Reverse Engineering Heterogeneous Applications

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Fabrizio Perin
von Italien

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz

Institut für Informatik und angewandte Mathematik

Reverse Engineering Heterogeneous Applications

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Fabrizio Perin
von Italien

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz

Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 22. November, 2012

Der Dekan:

Prof. Dr. Silvio Decurtins

This dissertation is available as a free download from scg.unibe.ch.

November, 2012.

Acknowledgements

First of all, I would like to thank Prof. Dr. Oscar Niestrasz for giving me the opportunity to work at the Software Composition Group and for his support throughout the years.

I would like to thank Dr. Leon Moonen for accepting to be on the PhD committee. I enjoyed the good discussions we had when we met at various reverse engineering conferences.

I thank Prof. Dr. Paolo Favaro for accepting to chair the examination.

I would like to thank all the former and current members of the Software Composition Group. It was a pleasure to work with you: Lukas Renggli, Jorge Ressia, Tudor Gîrba, Adrian Lienhard, Erwann Wernli, Mircea Lungu, Marcus Denker, David Röthlisberger, Niko Schwarz and Toon Verwaest. Thanks to Iris Keller who made the administrative work almost a pleasure and for the funny discussions we had.

I would like to express my gratitude to Tudor Gîrba for the inspiring discussions and for providing many of the ideas that have influenced this work.

Many thanks to my colleagues and friends Jorge Ressia and Lukas Renggli for their moral support and all the funny moment spent together.

Special thanks to my oldest and closest friend Stefano who knows me better than I know myself and who still stand me after over 20 years of friendship.

Thanks to Lea, Simone, Matteo, Giulia, Leo and Atsuko for being there all the times I needed some nice words and a beer.

Thanks a lot to the little Livia who is in this world since not much time but who already did a lot for me only with her presence.

Thanks to all my friends back in Italy and all the people I know that have supported me in this adventure. I cannot list you all here but you all know how valuable you are for me.

Above all, I would like to thank Emilia Fusi who changed everything ever since she entered my life.

If you don't mind, I will switch to Italian to thanks the persons who do not speak english but deserve to be mentioned more than others. Mille grazie alla mia famiglia che mi è sempre stata vicina e che si è sempre occupata di me anche dopo essermi trasferito in un'altro paese.

I almost forgot to thanks ImmigrationStopper and ByfrostTwister for the nice time spent gaming and talking about research.

Fabrizio Perin
November 22, 2012

Abstract

Nowadays a large majority of software systems are built using various technologies that in turn rely on different languages (e.g. Java, XML, SQL etc.). We call such systems heterogeneous applications (HAs). By contrast, we call software systems that are written in one language homogeneous applications. In HAs the information regarding the structure and the behaviour of the system is spread across various components and languages and the interactions between different application elements could be hidden. In this context applying existing reverse engineering and quality assurance techniques developed for homogeneous applications is not enough. These techniques have been created to measure quality or provide information about one aspect of the system and they cannot grasp the complexity of HAs.

In this dissertation we present our approach to support the analysis and evolution of HAs based on: (1) a unified first-class description of HAs and, (2) a meta-model that reifies the concept of horizontal and vertical dependencies between application elements at different levels of abstraction.

We implemented our approach in two tools, MooseEE and Carrack. The first is an extension of the Moose platform for software and data analysis and contains our unified meta-model for HAs. The latter is an engine to infer derived dependencies that can support the analysis of associations among the heterogeneous elements composing HA. We validate our approach and tools by case studies on industrial and open-source JEAs which demonstrate how we can handle the complexity of such applications and how we can solve problems deriving from their heterogeneous nature.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Most modern industrial strength software systems are built using various technologies and frameworks often relying on multiple languages (*e.g.*, Java, XML, SQL *etc.*). Due to this composition of elements the information regarding the structure and the behaviour of modern applications can be spread across various components. We call such systems "heterogeneous" and we define them as follows:

> **Heterogeneous Application**
> *A heterogeneous application (HA) is an application whose structural and behavioural information is encoded in several distinct languages or technological aspects.*

Enterprise applications (EAs) are a typical example of heterogeneous applications. The specific technological stack used to implement an EA depends on its constraints and functionalities. It is possible to identify, however, several common features shared by most EAs [Fowler, 2005a]:

- The need of persistent data. For this purpose EAs generally rely on databases.
- The use of complex, often web based, user interfaces to access the application functions and data.
- A complex business logic potentially developed in more than one language.

Several problems that occur in the development and in the analysis of HAs can involve more than one of these features, as well as other elements of the application, at a time.

A clarifying example problem, often encountered in the development of Java Enterprise applications (JEAs), is the identification of application transaction scopes. Application transactions have been defined in JEA to guarantee atomicity of operations and to provide isolation to services accessed by different clients. It is important to ensure that critical services are properly contained within their own transaction

scopes, while starting unnecessary transactions should be avoided for performance reasons. In JEAs a method can be defined as part of a transaction by setting certain attributes in a XML configuration file called deployment descriptor. To analyze the application transaction scope in JEAs we need to combine knowledge contained in the Java source code with the one contained in the application descriptor files.

Another example comes from the analysis of data-intensive systems [Cleve *et al.*, 2010b]. Such systems generally comprise a database and a collection of application programs in strong interaction with the former. The evolution of data-intensive systems implies the co-evolution of the application and its database together with their interconnections. Due to the elevated costs of data reengineering, however, the focus of the evolution process shifts on the improvement of an application, while leaving its database untouched. Data reengineering would benefit from instruments that can perform analyses on both the application database and source code.

These examples demonstrate how several problems concerning heterogeneous applications can only be solved by taking into account more than one part of the application at a time.

We list below the aspects that need to be considered when dealing with heterogeneous systems and that represent the main challenges encountered in HAs analysis:

Languages. The application source code used to implement the application logic is central to a number of analyses [Binkley, 2007]. It is not always straightforward, however, to apply these analyses in many real case scenarios. As software systems become more interoperable, in fact, the application logic can be implemented with multiple languages simultaneously. Linos *et al.* reported that in 1998 one third of the software developed in the US was written using two languages and approximately 10% with three or more [Linos *et al.*, 2003]. To understand structure and behaviour of HAs we need to be able to deal with more than one language at a time.

Architecture. Software architecture is important to understand large applications and to support their evolution [Ducasse and Pollet, 2009]. The major challenges of *software architecture reconstruction* (SAR) are abstracting, identifying and presenting higher level *views* [IEEE, 2000] from lower level and heterogeneous information [Ducasse and Pollet, 2009]. By unifying a representation of application concrete architectures and a description of source code languages we can serve the purposes for which SAR techniques are used [Garlan, 2000]. Moreover, we can enable multiple views and validations on HAs' software and architecture [Perin *et al.*, 2010].

Persistency. Data reverse engineering is as important as reverse engineering the source code [Chikofsky, 1996]. A widespread technology to ensure data persistence is that of relational databases Enabling the combination of software and database analysis is crucial if we want to support the evolution of heterogeneous applications [Cleve *et al.*, 2010b].

Technologies. The specific technologies and frameworks used to implement an application are an invaluable source of information (*e.g.*, the build systems used to transform the source code into deployable components [Spinellis, 2008]). Understanding the impact these technologies have on the application could lead to a deeper understanding of the application structure and behaviour.

Dependencies. The connections between software elements are important to understand the behaviour of an application and to consequently perform tasks like impact analyses [Bohner and Arnold, 1996] and change propagation [Rajlich, 1997]. Connections among software components are often not directly encoded in the software but can be derived by other connections [Zhifeng Yu, 2001; Vanciu and Rajlich, 2010]. In EAs some connections can be derived from relationships between elements in a different domain. For example connections at source code level can be used to link user interface components [Aryani *et al.*, 2011]. To understand the nature of derived connections between HAs elements, we need to make them explicit. In addition, we also need to expose the information used to derive these connections and make it accessible at multiple levels of abstraction.

Software analysis techniques and tools must take these aspects into account to support the maintenance and the evolution of HAs.

Few generic approaches and techniques capable of dealing with the complexity and heterogeneity of modern software systems have been proposed in the literature.

Among those, GUPRO (Generic Understanding of PROgrams) is *"an integrated workbench to support program understanding of heterogeneous software systems on arbitrary levels of granularity"* [Ebert *et al.*, 2002] which implements the EER/GRAL approach proposed by Kullbach *et al.* [Kullbach *et al.*, 1998]. This tool is designed to handle the multi-language nature of heterogeneous applications. Its focus, however, is on analyzing application source codes and not on the modelling of HAs in all their parts.

Marinescu *et al.* recognized the need to model heterogeneous applications as a whole even though their meta-model [Marinescu and Jurca, 2006] focuses only on relational databases and so do their analyses [Marinescu, 2007b; Marinescu, 2007a].

MoDISCO[1] is an Eclipse plug-in meant to support the modernization of legacy HAs. The modernization process implemented in MoDISCO comprises an initial understanding phase based on a model of the legacy system under analysis. This model can also be exploited for purposes other than software modernization, as, for example, software re-documentation or software visualizations. MoDISCO also provides the user with the possibility to define new models. The meta modeling facility of MoDISCO is based on two OMG standards: the Knowledge Discovery Metamodel[2] (KDM) and the Software Metrics Metamodel[3] (SMM).

KDM specifies a comprehensive set of meta-models that describe existing software systems in preparation for software assurance and modernization. The KDM specification contains twelve packages that describe different aspects of modern software systems. KDM would be a perfect candidate to perform the same implementations and analyses described in this dissertation, however, we argue that a simpler model which is easier to extend is still required to support analyses of HAs.

On the side of software dependency analyses several approaches have been proposed to trace dependencies between software elements using information from different domains.

Structural coupling metrics have received increasing attention in the past years resulting in many different approaches ranging from dynamic coupling [Arisholm *et al.*, 2004; Hassoun *et al.*, 2004] to evolutionary and logical coupling [Zimmermann *et al.*, 2004; Gall *et al.*, 2003]. Other coupling metrics have been proposed based on the concepts specific to software systems [Poshyvanyk *et al.*, 2006; Poshyvanyk and Marcus, 2006; Poshyvanyk *et al.*, 2009; Gethers and Poshyvanyk, 2010]. These approaches measure the relations among software entities by considering latent topics from the source code. Gall *et al.* have shown that semantic metrics computed from design documents correlate well with semantic metrics computed from the source code and could be used as proxies for them [Gall *et al.*, 2008].

An alternative approach to source code analysis is mining dependencies from software repositories [Ying *et al.*, 2004; Hindle and Jordan, 2004; Walker *et al.*, 2006; Kagdi *et al.*, 2007; D'Ambros *et al.*, 2009]. Although the information is gathered from different domains the presented approaches focus on analysis of dependencies at source code level.

None of the previously mentioned approaches provide a generic and extensible solution to support the analysis of dependencies between heterogeneous elements at multiple levels of abstraction. KDM provides some support for heterogeneous dependency analysis, however, does not provide a generic solution.

1 http://www.eclipse.org/MoDisco/

2 http://www.omg.org/spec/KDM/1.3/

3 http://www.omg.org/spec/SMM/

We state our thesis as follows:

> **Thesis**
> *To support the analysis and the evolution of heterogeneous applications: (1) we need a homogeneous first-class representation of the heterogeneous application components and, (2) we need to expose the direct and indirect relationships among the application elements belonging to different domains at multiple levels of abstraction.*

We argue that a homogeneous representation of the elements composing heterogeneous applications at different levels of abstraction would enable different kinds of analyses based on software metrics, software visualizations and queries. We also argue that the implicit dependencies between the semantically different elements composing HAs need to be reified to support tasks like impact change propagation and software understanding in modern software systems.

In this dissertation we present our first-class representation to deal with three aspects of HAs:

Architecture. Architecture is usually represented as a set of components and connectors among these components [Shaw and Garlan, 1996]. Such a representation is hardly detailed enough to describe the concrete architecture of HAs. Our meta-model for software architectures mixes a representation of components and connectors with a description of architectural layers [Fowler, 2005a].

Persistency. Database management systems (DBMSs) often have a separate life cycle from the applications accessing them due to the complexity and the elevated costs of data reverse engineering. Our approach comprises a first class description of relational databases to enable analyses that can support the coevolution of HAs source code and persistent data models.

Technologies. Since its introduction in 1999, Java 2 Platform Enterprise Edition (J2EE) became one of the standard technologies for enterprise application development. Because JEAs are so extensively used we reify the elements of a JEA specific technology such as Enterprise Java Beans (EJBs). This description can enable various analyses which aim to understand the impact this technology has on the system.

Each of these descriptions is explicitly connected to a meta-model for object-oriented and procedural languages capable of describing most of the languages used in the development of HAs. These descriptions can be used independently to deal with specific aspects of HAs or can be combined to deal with orthogonal problems which cover more than one aspect at a time. We implemented our approach in a tool

called MooseEE as an extension of the Moose platform for data and software analysis [Nierstrasz *et al.*, 2005]. Moose includes the FAMIX meta-model for object-oriented and procedural languages [Tichelaar *et al.*, 2000] which we use to model HAs source code.

Figure 1.1 depicts the three first-class representations we discuss in this dissertation and indicates the chapters in which the respective parts are validated.

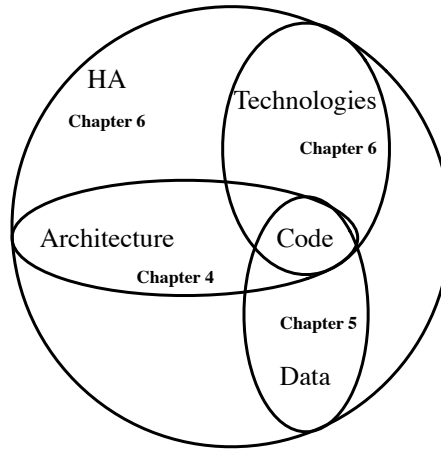


Figure 1.1: Points of view considered in the analysis of heterogeneous applications

In this dissertation we also present our approach to support dependency analyses in HAs. Some of the connections between HA elements are implicit and can be derived from relationships between elements in different domains [Aryani *et al.*, 2011]. To support the analysis of derived dependencies in HAs we reify the concept of vertical and horizontal relationship applied to graph nodes. Vertical relationships are used to move from a lower to a higher level of abstraction, while horizontal relationships are use to move between elements at a similar level of abstraction. The semantic of these relationships is not strict and can be adapted to follow the user needs. The resulting first-class description can then be used to infer dependencies between the semantically different elements of our homogeneous description of HAs as well as in any other graph-based structure. We implemented this approach in the Carrack derived dependencies inference engine. Carrack allows the user to script the definition of the desired derived dependencies on top of a given model.

The following list details the contributions with some extended case studies, which serve as the validation of our approach:

Architectural Validation. We validate our architectural representation of HAs by validating architectural constraints and recovering architecture of an industrial JEA. The recovery of the system's concrete architecture is achieved by merging information from the source code with the expertise of the application developers following a bottom-up approach [Ducasse and Pollet, 2009].

Conceptual Schema Recovery. We applied our meta-model for relational databases to support the understanding and the evolution of a data-intensive open-source content management system [Cleve *et al.*, 2010b]. By merging information from the source code and the database structure to detect object-relational persistence patterns we built a conceptual schema of the software under analysis. The conceptual schema provides an overview on the system that can support the co-evolution of the database and the application source code.

Transaction Flow. Application transactions have been defined in JEAs to guarantee atomicity of operations and to provide a certain level of isolation of services accessed by different clients. By merging information about EJBs and the source code, we were able to recover and visualize the transaction's scope in the JEA source code. By exploiting the architectural model and the relational databases analysis implementation included in our approach we could also identify: architectural violations of the elements involved in a transaction as well as unsafe accesses to the database from methods outside a transaction scope.

Dependency analysis in HAs. Our approach for the identification and exploitation of derived dependencies in HAs as been validated on the same case study used to validate our architectural model. Architectural level associations are typically, but not strictly, derived from source code level associations. By exploiting our approach we demonstrate how we could generalize the process to infer dependencies at the architectural level from the dependencies between source code entities. We also demonstrate that our approach to derive dependencies is general enough that it can be applied on generic graph-based structures.

Outline

This dissertation is structured as follows:

Chapter 2 discusses the related work of this thesis. We present a review summary of various solutions in the literature to address the analysis of different aspects of heterogeneous applications and we identify the difference with our work.

Chapter 3 presents our approach to deal with HA analysis based on a first-class description of HA architectural elements, relational databases and JEA technology such as EJBs.

Chapter 4 demonstrates the use of our architectural description to perform architectural understanding and validation tasks.

Chapter 5 evaluates our relational database model to enable the co-evolution of the source code and the database.

Chapter 6 describes how a homogeneous representation of HAs can lead to solving an orthogonal problem such as the application transaction scope identification in JEAs in all its facets.

Chapter 7 describes our approach to identifying and exploiting derived dependencies between HA elements and graph-based models. The case study we used in this chapter is the same used for the validation of our architectural model.

Chapter 8 concludes the dissertation and outlines future work.

Chapter 2

Heterogeneous Application Analysis

In the literature few approaches have been proposed to deal with HAs as a whole. On the other hand, a number of techniques and tools have been proposed to analyze specific aspects of heterogeneous applications. In this chapter we focused our attention on those approaches that support the analysis of multi-language systems (Section 2.1), persistent data structures (Section 2.2) and architectures (Section 2.3).

Software dependency analysis is the process of finding the connection between software components. Knowledge of software dependencies is vital to many change impact analysis methods and other maintenance activities [Rajlich, 1997; Zhifeng Yu, 2001; Hassan and Holt, 2004; Vanciu and Rajlich, 2010]. HAs are implemented using multiple languages and various technologies and frameworks. Instruments that take into account this composition of elements are needed to support dependency analysis in HAs. In this chapter we also summarize the state of the art in dependency analysis approaches and tools.

2.1 Multi-Language Source Code Analysis

As software systems become more interoperable, it is common to see hybrid systems composed of multiple programming languages (*e.g.*, C++ and Python). Already in 1998 it is reported that one third of the software developed in the US was written using two languages and around 10% with three or more languages [Linos *et al.*, 2003]. Heterogeneous applications are a good example of such multi-language systems. Several approaches have been proposed to support program understanding of multi-language applications.

Kullbach *et al.* presented a graph-based conceptual modeling approach [Kullbach *et al.*, 1998] enabled by an extended entity-relationship dialect (EER) and the GRAL constraint language [Ebert *et al.*, 1996]. In this approach each source code language to analyze has a conceptual model describing the language in term of entities and

relationships between these entities. The various conceptual models are then integrated within a unified conceptual model which defines the relationships between the entities of the multiple languages implemented. Based on the EER/GRAL approach Ebert *et al.* presented GUPRO (Generic Understanding of PROgrams) which is *"an integrated workbench to support program understanding of heterogeneous software systems on arbitrary levels of granularity"* [Ebert *et al.*, 2002]. GUPRO is able to handle the multi-language nature of heterogeneous applications, however, its application is limited to the analysis of application source code. For each new language we want to analyze with GUPRO we need to define both the language description and the relationships with the other language descriptions. This workflow leads to the definition of all the required connections, thus making the multi-language representation in GUPRO complicated to understand and modify. The technology used to represent program information implies the analysis mechanism [Kullbach *et al.*, 1998]. GUPRO uses GReQL (Graph Repository Query Language) to query its graph-based program descriptions [Ebert *et al.*, 2002]. Even though GReQL queries are reusable because they can be applied to any graph structure, they can also became quite big and difficult to implement for some validations.

Instead of creating their own representation for multi-language applications, Linos *et al.* proposed a set of multi-language software metrics based on the analysis of the *Microsoft Intermediate Language* (MSIL) [Linos *et al.*, 2007]. Linos *et al.* demonstrates that the analysis of the MSIL is as effective as when the same analysis is conducted on the original language. This approach has the same advantages of all the approaches based on an intermediate representation, however, the drawback is that it is not extensible and it works on the languages supported by the .NET environment only.

All the approaches presented until now are based on models of specific languages or groups of languages. Strein *et al.* tried to take a step further proposing an approach for multi-language source code refactoring and analysis based on a language independent meta-model [Strein *et al.*, 2006]. For language independence the authors designed an abstract representation able to represent any language in any paradigm. The aim of Strein *et al.* is to provide a generic language description to support multi-language refactoring. These properties are fulfilled using a three level abstract representation composed of a model, a meta-model and a meta-meta-model [Strein *et al.*, 2006]. Strain *et al.* implemented their approach in a tool called *X-Develop*. Each language supported by this tool is represented by a so called *front-end* which is transformed to fit a multi-language model. The information loss caused by the high level common representation does not affect the refactoring abilities of X-Develop, however, we believe such abstract representation restricts the possible software analysis that could be done on the source code.

2.2 Data-Intensive Systems Analysis

Data Reverse Engineering is a specific information system engineering domain aiming at rebuilding the documentation of legacy databases [Hainaut *et al.*, 2000]. Data reverse engineering aims to address specific problems like: (i) the weakness of the DBMS technical models that cannot express all the constructs and constraints represented in conceptual schemas, (ii) the weakness of databases designed by not skilled enough software designers, (iii) the obsolescence of data structures and, (iv) the lack of design documentation [Hainaut *et al.*, 2000].

Interest for data reverse engineering is increasing for businesses and organizations currently facing the critical issues of managing large amounts of data [Tallon, 2010], *e.g.*, related on one hand to the internal diffusion of data warehouses and analytics for strategic decision support systems [Chikofsky, 1996], on the other hand to the growth of data coming from external sources like social networks and data services [Garcia-Molina *et al.*, 2011; Carey *et al.*, 2012].

The understanding of data structures and of programs that manipulate them are strictly tied to provide the full functional specifications of an information system [Hainaut *et al.*, 2000]. Various strategies and methodologies for data reverse engineering have been proposed and discussed in literature [Hainaut *et al.*, 2000; Mian and Hussain, 2008]. As for the available tools Davis *et al.* [Davis and Aiken, 2000] shows the relevance of well known data modelers like CA ERwin Data Modeler or Embarcadero ER/Studio, challenged more by suite products from other major vendors such as IBM or Microsoft [Hammond *et al.*, 2008] with respect to open source solutions or ontology based techniques.

Methodologies or approaches providing a unified perspective have been poorly investigated in the literature. Early attempts tried to define strategies at schema, data, and program level to migrate data-intensive applications from legacy data management systems [Henrard *et al.*, 2002].

Cleve *et al.* [Cleve *et al.*, 2010a] proposed a comprehensive approach for the rapid development and evolution of data-intensive applications. This approach combines the automated creation of a relational database from a conceptual schema and the automated generation of a data manipulation API on the created relational database.

Henrard *et al.* [Henrard *et al.*, 2007] proposed methodologies, techniques and tools to analyze and to adapt legacy data-intensive programs in support to database reverse engineering with particular attention to the problem of database platform migration.

Marinescu *et al.* proposed a meta-model for enterprise applications which was only focused on object-oriented software systems and relational databases [Marinescu and Jurca, 2006]. This description has been used to address various problems: With the purpose of improving the quality of the data structure Marinescu proposed an approach to identify the relational discrepancies between database schemas and source code in EAs [Marinescu, 2007b]. To support the evolution of database schemas Marinescu described how to enrich the semantic of the foreign key constraints defined between database tables looking at the relationships between source code elements [Marinescu, 2007a]. Marinescu proposed also a set of rules for the identification of the Fowler's enterprise patterns defined for data management [Fowler, 2005a] by leveraging her description of EAs [Marinescu, 2006].

Design flaws in EAs sometime depend on the bad design of the interaction between the source code and the database. Keller investigated the object to tables mapping problem and he described patterns to support the design of applications' access layer [Keller, 1997; Keller, 1998].

2.3 Software Architecture Reconstruction and Validation

Software architecture has been defined by several people with little agreement. Shaw and Garlan [Shaw and Garlan, 1996] defined software architecture in terms of *components* and *connectors* that mediate the interaction between the components. This definition resemble the one proposed by IEEE [IEEE, 2000]. Components can represent clients and servers, databases and computational elements in general. Connectors can be procedure calls, shared variable accesses or more complex elements like client-server protocols. This representation is generic and so can be used to describe a variety of software architectures. One drawback is that this representation is focused on technological aspects only. The point of view of the technology, however, is not the only one. Aspects related to the system domain or the business context of an application must be taken into account to provide all the stakeholders with a complete picture of a software system architecture [Medvidovic *et al.*, 2007]. Another drawback of such architectural description is that components and connectors have a polymorphic semantic which must be specified in an unambiguous manner to achieve precision in architectural analyses and architectural understanding.

Software architectures can be derived from software documentation or human knowledge or they can be reconstructed from the source code. In the first case we talk about conceptual architecture, in the latter we talk about concrete architecture. Software architecture reconstruction (SAR) can be achieved by following

top-down or bottom-up approaches [Ducasse and Pollet, 2009] and it is done for several purposes: Understanding, reuse, construction, evolution, analysis and management [Garlan, 2000].

Several approaches have been proposed over the years to support SAR. Ducasse *et al.* summarize them from the point of view of the goal and the process they use to reconstruct the architecture of a system [Ducasse and Pollet, 2009]. Approaches not included in the surveys of Ducasse *et al.* have been discussed by Knodel *et al.* which organized the state of the art considering the method used by the tools to perform architectural compliance checking [Knodel and Popescu, 2007].

Independently from the purpose for which SAR is done all the approaches we encountered describe architectures only in terms of components and connectors. We argue that this definition is too simplistic to express the complexity of heterogeneous application architectures. On the one hand, we need to identify and reify more concrete elements to represent the structure of HA architectures. On the other hand, we need to identify those elements that can represent the application domain and its business context to describe an architecture from the points of view of the various system stakeholders.

In this dissertation, however, we focus our attention on identifying and reifying those architectural elements that are part of the structure of HAs . We reified more specific architectural elements like, for example, architectural layers [Fowler, 2005a] to support effective architectural understanding and validation on HA.

2.4 Dependency Analysis

When software maintainers change a software entity, they have to search for other related entities and update them accordingly. This is not a trivial task, and many bugs are introduced by programmers who fail to properly propagate changes [Hassan and Holt, 2006]. Knowledge of software dependencies is vital to many change impact analysis methods and other maintenance activities [Zhifeng Yu, 2001; Vanciu and Rajlich, 2010; Rajlich, 1997; Hassan and Holt, 2004]. Source code analysis can be used to trace dependencies [Binkley, 2007]; however, it is not an easy approach to apply to EAs due to their heterogeneous composition of languages and frameworks.

In the literature, several formal models of change propagation have been introduced. Luqi [Luqi, 1990] presented a graph model for software evolution based on indirect relationships between components. Rajlich [Rajlich, 1997] introduced a model for change propagation based on graph rewriting which requires

an understanding of the dependencies between software elements. Arnold and Bohner [Bohner and Arnold, 1996] model change impact analysis as a cycle of revisions derived from relationships between software elements. Mirarab *et al.* [Mirarab *et al.*, 2007] introduced a hybrid impact analysis method based on dependency information and co-change history. The knowledge of software dependencies is the prerequisite for these impact analysis models. The other key applications of dependency analysis are program comprehension, concept location and reverse engineering [Cleary and Exton, 2007; Tzerpos and Holt, 2000; Walker *et al.*, 2006; Marinescu, 2007a].

Source code analysis [Binkley, 2007] is an established approach for tracing software dependencies [Harman *et al.*, 2009; Cleve *et al.*, 2006] or evaluating the evolution of code and design [Hammad *et al.*, 2009]. One of the best-known code analysis methods is *program slicing*, which has been exhaustively explored by many researchers and extended to many programming paradigms [Binkley and Harman, 2004; Willmor *et al.*, 2004; Xu *et al.*, 2005; Silva, 2011]. Source code analysis is further enhanced using dynamic analysis [Xiao and Tzerpos, 2005; Cornelissen *et al.*, 2009] to capture dependencies which might not be traceable from static relationships between software elements.

Structural coupling metrics have received a lot of attention in the past years resulting in many different approaches ranging from dynamic coupling [Arisholm *et al.*, 2004; Hassoun *et al.*, 2004] to evolutionary and logical coupling [Zimmermann *et al.*, 2004; Gall *et al.*, 2003]. Metrics like Coupling Between Objects (CBO) or CBO' [Chidamber and Kemerer, 1994] consider the inheritance between classes to measure the coupling among software elements. Other metrics like Response For Class (RFC) [Chidamber and Kemerer, 1991] and RFC_∞ [Chidamber and Kemerer, 1994] consider indirect relations among classes based on a level of indirection in the invocation chain of the class methods.

More recent research effort has concentrated on defining coupling metrics based on the concepts specific to software systems [Poshyvanyk *et al.*, 2006; Poshyvanyk and Marcus, 2006; Poshyvanyk *et al.*, 2009; Gethers and Poshyvanyk, 2010]. These approaches attempt to identify and measure the relation among software entities in object-oriented software by considering latent topics from the source code.

An alternative approach to source code analysis is mining dependencies from software repositories [Ying *et al.*, 2004; Hindle and Jordan, 2004; D'Ambros *et al.*, 2009; Kagdi *et al.*, 2007; Walker *et al.*, 2006]. It can be argued that these approaches are less expensive, and require less technical expertise. On the other hand, they are not applicable where maintenance history is not accessible.

2.5 Heterogeneous Application Analyses

Few generic approaches and techniques capable of dealing with the complexity and heterogeneity of modern software systems have been proposed. The EER/GRAL approach [Kullbach *et al.*, 1998] implemented in GUPRO [Ebert *et al.*, 2002] can actually be used to perform architectural constraint validation although its focus is the analysis of multi-language systems. Marinescu *et al.* recognized the need for an approach to deal with heterogeneous applications as a whole even though their meta-model [Marinescu and Jurca, 2006] focuses only on relational databases and so do their analyses [Marinescu, 2007b; Marinescu, 2007a].

A well known tool capable to support the analysis of HAs is an Eclipse plug-in called MoDISCO[1]. The main purpose of this tool is to support the modernization of legacy heterogeneous software systems. The modernization process implemented in MoDISCO comprises an initial understanding phase based on a model of the legacy system under analysis. The process we use to support the analysis of heterogeneous applications is similar to the one implemented in MoDisco but its purpose is to support the understanding of heterogeneous applications. The meta modeling facility of MoDISCO is based on two OMG standards: the Knowledge Discovery Metamodel[2] (KDM) and the Software Metrics Metamodel[3] (SMM).

KDM specifies a comprehensive set of meta-models that describe existing software systems in preparation for software assurance and modernization. The KDM specification contains twelve packages that describe different aspects of modern software systems. Each of these packages is defined by one or more class diagrams. The core package of KDM is a specification of the basic abstractions of KDM which serves as a base for the other KDM packages and that defines elements like entities, relationships, container hierarchies, *etc.*

Similarly to KDM also the Moose platform, which we extended to implement our approach, has a self-described core model that has been inspired by the Essential Meta Object Facility (EMOF) [Group, 2004]. This meta-meta-model is called FAME and can be used to ensure interoperability between tools. On top of this meta-meta-model has been created a family of meta-models to describe different aspects of modern software systems. These meta-models are typically geared towards enabling analysis and providing a rich API that can be used for querying and navigation. In addition, around the Moose platform several tools for scripting software visualizations, reports, browsers *etc.* have been developed.

1 http://www.eclipse.org/MoDisco/

2 http://www.omg.org/spec/KDM/1.3/

3 http://www.omg.org/spec/SMM/

KDM would be a perfect candidate to perform the same implementations and analyses described in this dissertation, however, we decided to base our work on Moose and to extend this platform rather than KDM. This choice is justified by the previous experience we have with Moose. Another aspect we considered is that Moose is implemented in Smalltalk. Because we can use Smalltalk as a scripting language, the APIs defined by the meta-models can be used as query languages.

With the intent to support the modernization of legacy enterprise systems the OMG Architecture-Driven Modernization (ADM) Task Force based on the KDM model a modernization process described in detail by Ulrich and Newcomb [Ulrich and Newcomb, 2010] in their book. The modernization process consists in an initial ascendent phase of analysis and modeling and a descendent transformation phase. Both these phases touch the physical, the logical and the business level of the application to modernize. The authors used this process on several industrial strength case studies to prove its applicability. The process we use in Moose focuses on the analysis and understanding of heterogeneous systems, however, the process defined by the ADM Task Force could be fully implemented in Moose.

All the approaches presented in Section 2.4 focus on the analysis of the dependencies at source code level. Few approaches attempt to trace dependencies between source code elements by taking into account information from different domains other than the source code [Cleve *et al.*, 2006; Kagdi *et al.*, 2007; Gethers and Poshyvanyk, 2010; Vanciu and Rajlich, 2010; Aryani *et al.*, 2011]. Still, the purpose of these approaches is to understand the connections between source code elements.

Yazdanshenas and Moonen [Yazdanshenas and Moonen, 2011] proposed an approach to analyze the behavior of heterogeneous systems by analyzing their configuration files. This approach comprises a model of the heterogeneous system based on the KDM model which is populated with Component Dependence Graphs (CDGs) and Inter-Component Dependence Graphs (ICDGs) information extracted by using program slicing techniques [Horwitz *et al.*, 1990]. Configuration files are indeed a vital information source if we want to understand the behavior of HAs, also, the approach proposed by Yazdanshenas and Moonen is software independent since it is based on an extension of the KDM model.

On top of this approach Yazdanshenas and Moonen built an impact change propagation technique to estimate the ripple effects of changes within component-based product families [Yazdanshenas and Moonen, 2012a]. One interesting outcome of this approach is the possibility to move from fine-grained to coarse-grained dependencies and vice versa. This is an important requirement in the analysis of dependencies between the heterogeneous elements composing modern software systems. Still based on the same KDM model extension [Yazdanshenas and Moonen, 2011],

Yazdanshenas and Moonen proposed also a set of views that represent system-wide information flows at various levels of abstraction [Yazdanshenas and Moonen, 2012b]. These works from Yazdanshenas and Moonen offer a further demonstration that the KDM meta-model is a good candidate to be extended to perform structural and behavioral analyses on heterogeneous applications.

We argue that prebuilt analyses most of the time fail in addressing the real problems encountered in the analyses of existing applications. Therefore our intent is to propose a compact, easy to understand and use meta-model to describe the elements composing heterogeneous applications together with their connections.

All the approaches mentioned in this section are system independent, however, they do not support generically the analysis of dependencies derived by other kind of dependencies. Yazdanshenas and Moonen actually inject the elements of interest from their KDM model into a generic graph representation which is then queried to implement their information flow analysis. However, they do not classify edges as horizontal and vertical, so they do not have an automatic inference of derived relations as the one we propose in this dissertation.

Therefore, we argue that techniques and tools that can generically support dependency analysis involving all the elements composing HAs are still needed.

Chapter 3

Enabling Heterogeneous Application Analysis

In this chapter we present our approach to support the analysis and evolution of HAs based on a unified meta-model describing several elements that comprise these applications. In particular, we deal with three aspects of HAs: architecture, persistency and technologies.

To serve the purposes for which SAR techniques are used [Garlan, 2000] we provide a first-class description of software architectures which combines a representation of components and connectors with one describing architectural layers [Fowler, 2005a]. To support the co-evolution of a HA's source code and its persistent data model we define a meta-model for relational databases and we specify the relationships between the relational elements and the source code elements. To enable the analysis of JEA specific technologies we provide a first-class description of Enterprise Java Beans (EJBs).

These descriptions can be used independently to deal with specific aspects of HAs or can be combined to deal with orthogonal problems which cover more than one aspect at a time. We implemented our first-class description for HAs in a tool called MooseEE which is an extension of the Moose platform for data and software analysis [Nierstrasz *et al.*, 2005]. A brief description of the Moose platform is provided in Appendix B.

3.1 Representing Architectural Elements

Architecture is usually described in terms of a set of components and connectors among these components [Shaw and Garlan, 1996]. This model is too abstract and needs to be specialized all the time we need to represent a specific architecture.

The polymorphic semantic of components and connectors could also lead to confusions and misunderstandings if not properly defined. Finally, components and connectors might be an excellent common ground for structural definitions of architectures, however, they fail in describing other aspects like the domain or the business context of applications. We argue that reasoning in term of components and connectors it is not enough to have a complete picture on HAs. In this dissertation we focus on defining a more detailed structural model of HA architectures. Fowler [Fowler, 2005a] argues that layering is one of the most common techniques used by software designers to decompose a software system into parts. By merging the architectural description of Shaw and Garland [Shaw and Garlan, 1996] and Fowler [Fowler, 2005a] we define a generic first-class representation of architectural components and layers to support HAs architecture analyses. Our meta-model is shown in Figure 3.1.

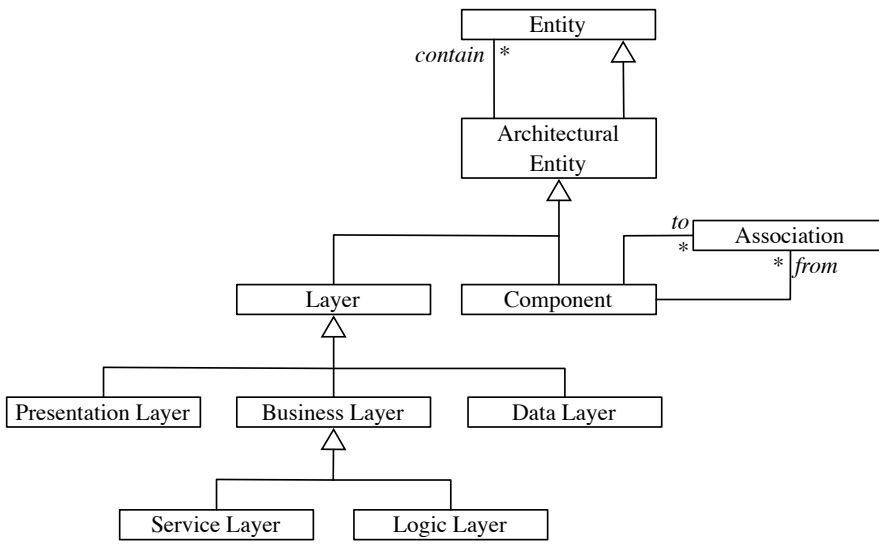


Figure 3.1: Meta-model of architectural elements in MooseEE

The most abstract element describing software architecture entities is `ArchitecturalEntity`. Its direct subclass, `Component`, is used to describe generic elements composing a software system. Components can be connected to one or more other components through the entity `Association`. The semantics of `Association` are not defined and they are specified by the user. This component-association architectural description is generic enough to provide a good extension point to add more specific architectural components.

Our overview of HA architectures includes also a description of software layers as shown in Figure 3.1. The `DataLayer` contains those software elements which access or represent data, usually persistent, within the software system. The `PresentationLayer` contains all the elements which define the application user interface (*e.g.*, HTML pages or Java classes implementing a rich-client UI). The `BusinessLayer` includes the application logic and it is organized into two sublayers namely `ServiceLayer` and `LogicLayer`. The first contains the software elements used as entry points by the application UI, while the latter contains the software elements actually implementing the application logic. Since several layering definitions have been proposed [Fowler, 2005b] the semantics of the different layers can vary accordingly to the layering strategy in place. For example, accordingly to the definition of Alur *et al.* [Alur *et al.*, 2001], the presentation layer contains the UI elements that run on the server side. Alur *et al.* define a new layer for J2EE applications called *Client* that contains the UI elements running on the client side.

Since architectural views are used to describe application elements that are not necessarily source code entities, an `ArchitecturalEntity` can *contain* zero or more instances of `entity`. This means that architectural elements can contain other architectural entities.

3.2 Relational Database Description

Relational databases are extensively used in the development of EAs due to their need of persistent data. Figure 3.2 shows our first-class description for relational databases.

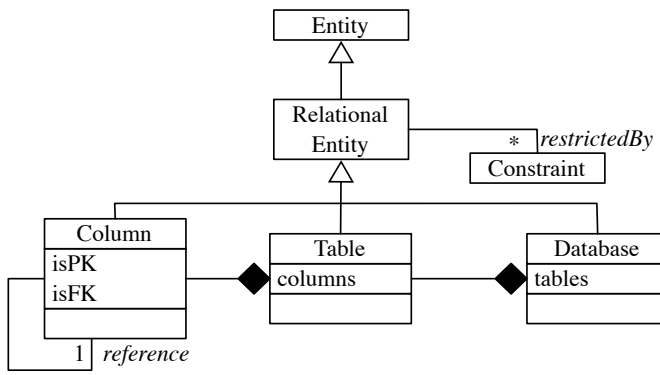


Figure 3.2: First-class description of relational databases

The entities modelling relational databases are self explanatory and they all subclass the generic element `RelationalEntity`. Database tables cannot exist outside the database containing them, similarly, table columns cannot exist outside the scope of the table they belong to. The relation *reference* represents connections among table columns established using foreign keys. Any relational entity can also have one or more constraints.

In Figure 3.3 we show the existing connections between relational elements and software entities. The elements `BehaviouralEntity` and `SourceEntity` are part of the FAMIX meta-model core. `BehaviouralEntity` is an abstract superclass for any software entity with a behaviour (*e.g.*, functions and methods). `SourceEntity` models FAMIX entities related to source code elements. The full FAMIX meta-model is shown in Figure B.2. The relationship *map* connects software entities describing relational entities at the source code level. Class-to-table or class attribute-to-column mappings are in our experience the most common ones in systems that use relational databases. The rationals of the map association can of course change to suit different kinds of mappings, such as, *e.g.*, the ones introduced by Keller or Fowler [Keller, 1997; Keller, 1998; Fowler, 2005a]. A representative example is a class implementing an active record pattern [Fowler, 2005a] that maps to the database entry it describes. The relation *access* represents read or write accesses from behavioural entities (*e.g.*, methods or functions) to relational entities. For example, a class method that reads the fields of a table.

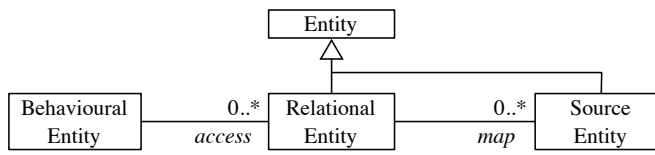


Figure 3.3: Connections between FAMIX meta-model entities for software systems and relational databases

3.3 Java Enterprise Applications Technologies

Technologies and frameworks can support developers in dealing with complexity. On the other hand, if not well managed they may turn into a burden, inflating development time and preventing the evolution of an application [Spinellis, 2008]. By reifying specific technological elements we can enable analyses to understand these technologies and how they affect a software system.

We considered in our work JEAs as instances of heterogeneous applications. Amongst the several technologies available for the development of JEAs, we focused on Enterprise Java Beans (EJBs) in both version 2.1 [DeMichiel, 2003] and 3.0 [Linda DeMichiel, 2006]. There are three types of Java beans: Session Beans are used to manage client connections in the server; Entity Beans are used to hold persistent data at the source code level while Message-Driven Beans are used to process messages asynchronously.

In Figure 3.4 we show our first-class representation of EJBs. The three classes modelling the three bean types extend the generic entity `JavaBean`. Each Java bean is connected with its Java class implementing its behaviour. The entity `Class` is part of the FAMIX meta-model depicted in Figure B.2.

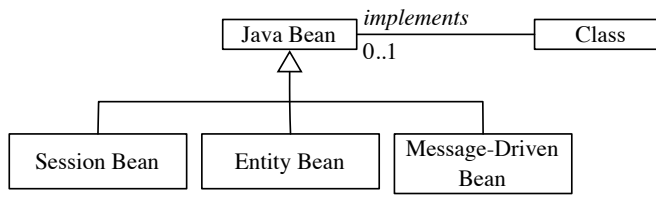


Figure 3.4: Enriched FAMIX Meta-Model (extensions are shown using bold)

3.4 Conclusions

In this chapter we presented our model for HAs which reifies: architectural elements, relational databases and JEAs' specific technologies. This unified description can be employed to analyze problems which involve various elements of an HA.

In the rest of this dissertation we will demonstrate the various parts of our homogeneous description of HAs independently on several case studies. In Chapter 4 we will describe how our description of architectural elements can be used to perform architectural understanding and validation on an industrial case study. In Chapter 5 we show how to support the evolution of a data-intensive system leveraging our meta-model of relational databases together with an *ad hoc* extension to describe conceptual schemas. Finally in Chapter 6 we use all the elements composing our meta-model for HAs to analyze application transaction scopes inside a JEA.

We implemented our model for HAs in a tool called MooseEE which is an extension of the Moose platform for software and data analysis. The Moose platform is briefly introduced in Appendix B.

Chapter 4

Architectural Understanding and Validation

In this chapter we present our approach to perform architectural understanding and validation tasks by exploiting our perspective on software architecture and source code (Figure 4.1).

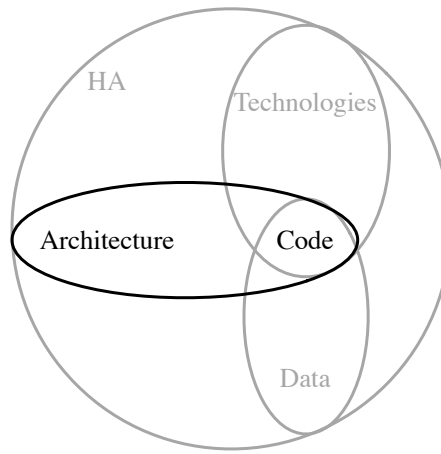


Figure 4.1: Architectural perspective on HAs

The application of choice for the validation of our approach is an industrial JEA that handles e-banking and online transactions. The architects of the banking system had two main requests for the analysis:

- an architectural overview of the distribution of the system components;
- a validation of the constraints on the connections between architectural elements.

The first step taken to address these requests was to adapt the architectural metamodel described in Section 3.1 to take into account the different component types that compose the banking system architecture. The second step was to design two polymetric visualizations to provide the architects with an overview on the system architecture from different points of view. Finally, we formalized several queries to validate architectural constraints on the associations between the architecture elements composing the system. The results of our analyses have been validated by the system architects who discovered problematic architectural elements they were not able to identify before.

4.1 Overview on Architectural Validation Techniques

Static approaches to architectural conformance checking can be categorized into three main groups: reflection models, relation conformance rules, and component access rules [Knodel and Popescu, 2007].

The reflexion model approach, introduced by Murphy [Murphy *et al.*, 2001] and extended by Koschke [Koschke and Simon, 2003], requires the user to define an abstract software architecture model. This model consists of architectural components mapped to concrete code entities and interconnect by semantically meaningful relationships. Source code dependencies are used to infer architectural level dependencies on the basis of suitable mappings provided by the user. The resulting dependencies are matched against the architectural relations defined by the user and the discrepancies are reported. Almost all the tools implementing this approach (Bauhaus [Raza *et al.*, 2006], Dependometer[1], SAVE [Stratton *et al.*, 2007; Knodel *et al.*, 2006], Sonargraph[2], Sotograph[3], Structure101[4]) provide support for a fixed set of programming languages and dependency types. ConQA [Deissenboeck *et al.*, 2005] is one of the few exceptions, as it supports the definition of new dependency types by allowing the user to customize the rules used to detect model discrepancies. We are not aware of any tool that relies on an easily extensible metamodel to express both architectural and source code elements. Moreover, all the cited tools rely on a predefined verification process and offer limited flexibility in modifying their analysis steps. Our goal is to provide an extensible architectural description suitable for representing the main elements of complex HAs, while also supporting software architecture reconstruction and validation tasks.

1 http://source.valtech.com/display/dpm/Dependometer
2 http://www.hello2morrow.com/products/sonargraph
3 http://www.hello2morrow.com/products/sotograph
4 http://www.headwaysoftware.com/products/index.php

The approaches based on relation conformance rules do not require one to specify an abstract architectural model. These approaches verify architectural constraints by using regular expressions directly on the source code elements. Most of the available tools that implement this approach (*i.e.*, IntensiVE [Mens and Kellens, 2006], dclcheck [Terra and Valente, 2009], Lattix LDM[5], .QL[6], tool by Eichberg et al. [Eichberg *et al.*, 2008]) are text-based and fully-declarative. IntensiVE and the tool developed by Eichberg et al. are both based on a logic programming language and are sufficiently expressive to support the definition of new checking rules. A special class of techniques based on relation conformance rules approach embeds dependency constraints directly into the source code (*e.g.*, ArchJava and ArchFace). The two main drawbacks of such techniques lie in their limited programming language support, and in their inherent unsuitability to deal with abstract entities.

Techniques based on component access rules define ports for each component and specify usage constraints for each port. This approach can be considered as a simplification of the previously described approach and was inspired by ports in ADL [Medvidovic and Taylor, 2000; Knodel and Popescu, 2007].

4.2 Case study

The software we analyzed is an industrial JEA responsible for managing e-banking and online transactions. This application has been online for several years and serves customers located around the world. The system has an HTML front end, a Java back-end and it uses an Oracle database to make data persist. The front-end is built using JavaServer Faces (JSF), the Java back-end is implemented using EJBs and the database is accessed directly via JDBC.

Figure 4.2 shows a meta-model level representation of the banking system architecture. All the architectural elements in the system are shown in this figure, however, the architectural configuration depicted is only one of the possible configurations these elements can have.

Conceptually the application is divided into components. A component can implement a function with a global or local scope. We will refer to them as "global" and "local" components, respectively. Local components with a regional scope (*e.g.*, EMEA) are called "global extensions". Each local component extends a global component to modify its behaviour. The system architects who commissioned this analysis need to monitor the size of these modifications in order to achieve automatic propagation of the changes made in the global components. Therefore, one of our

5 http://www.lattix.com/products/ldm-ldv

6 http://semmle.com/

analyses was focused on the identification of local components which customize big portions of the behaviour of the global components that they extend.

Each component can be part of an application layer. One or more components implement a function exposed to the end user. The architects refer to a function exposed by the banking system with the name of "functional component". Connections between functional components and between layers have been constrained by the architects in order to ensure isolation among the system functions and to have the correct flow of execution between layers.

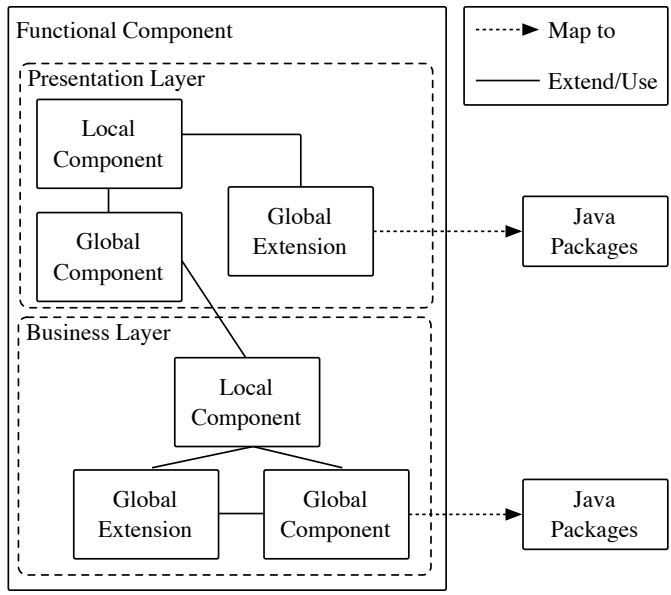


Figure 4.2: One of the possible architectural configuration that the system used in our case study can have. These figure includes meta-model level architectural components and connections.

Each component in our case study maps to a separate Java jar file that contains multiple Java packages. Relevant information is encoded in the jar file names: the name of the function implemented, the layer the jar file belongs to and the deployment localization (*i.e.*, global or local). We exploit this naming convention to recover the concrete architecture of the banking system following a bottom-up approach.

Table 4.1 shows some structural metrics of the industrial application under analysis. The size of the application chunk we analyzed was approximately $600,000$ lines of code. From this code we extracted 50 functional components and 101 local and global components and global extensions. In particular, we found 66 global

components and 28 local components. This means that approximately 40% of the global components have been extended and their behavior customized. This percentage is in general too high for the system architects, however, we will show in the following sections that most of these local components customize a small portion of the global components' behavior. The number of functional components and global extensions fulfilled the expectations of the architects.

Model Classes	3805
Lines of code	583182
Functional Components	50
Global Components	66
Local Components	28
Global Extensions	8

Table 4.1: Metrics describing the size of the banking system analyzed

4.3 Architectural Understanding and Validation

The architects managing the banking system had two main requests: (i) an overview of the distribution of the architectural components from the perspective of the application layers and the functional components; (ii) the identification of architectural constraint violations.

Figure 4.3 shows our extended architectural meta-model where the newly added architectural elements are depicted in bold. *SourceComponent* represents architectural components which are connected with source code elements and it is specialized by *GlobalComponent* and *LocalComponent*. The *LocalComponent* entity contains an attribute to distinguish between normal local components and global extensions. *FunctionalComponent* entities can contain one or more *SourceComponent*.

4.3.1 Architectural Understanding

The first issue we addressed was to provide the application architects with an effective way to understand how the components of the banking system were distributed from the point of view of the functional components and the application layers.

To address this issue we designed and developed two architectural polymetric visualizations, namely *Components per Function view* and *Components per Layer view*. Both these visualizations provide an overview of the application's architectural components highlighting their size and interactions and they show how the components

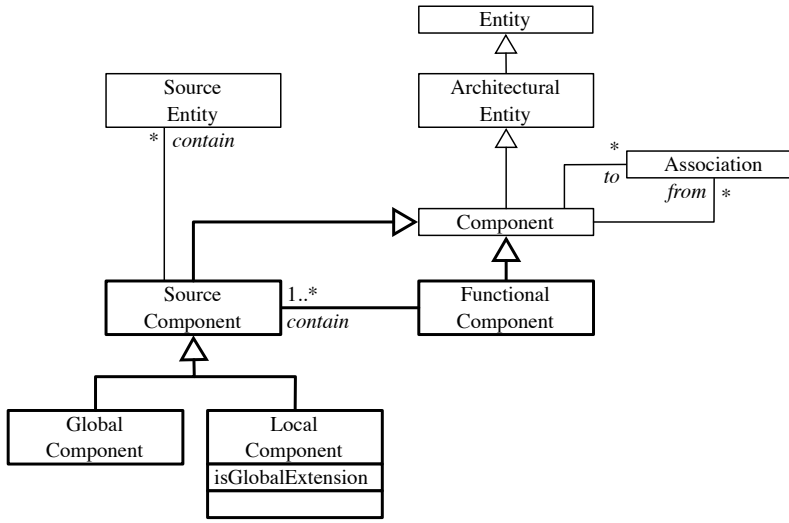


Figure 4.3: Meta-model of the architecture of the banking system

are distributed in the functional components and the application layers respectively. These visualizations, as any other visualization described in this dissertation, were developed using the Mondrian visualization engine [Meyer *et al.*, 2006].

Components per Function

The *Components per function view* makes it easy to identify complex components containing most of the function code. Complex local components should be avoided to reduce the application maintenance costs by achieving automatic propagation of the changes made in global components. Components that appear outside layers do not fully respect the naming convention defined on the component names. Functional components containing source components which belong to a single layer most certainly rely on external functions, therefore, they cannot be deployed on their own.

Figure 4.4 shows all the extracted architectural components split into functional components and layers. External rectangles represent functional components, while dashed rectangles represents the application layers. The source components are represented by squares. In Figure 4.4 the area of each square is proportional to the number of classes mapped by the corresponding component. The color gradient of the squares depends on the number of lines of code. The darker a component is the more lines of code it contains inside the scope of its functional component. Components with a blue border are local, those with a cyan border are global extensions while the others are global components. Components are represented as hierarchies

that highlight association order. This means that source entities contained in the components on top have a direct association (*e.g.*, derived by method invocations) to the elements below. Associations between components are represented by gray edges.

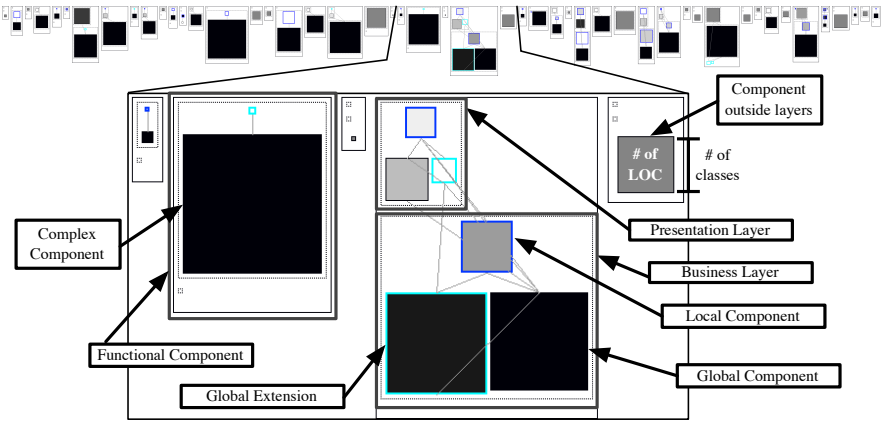


Figure 4.4: *Components per function view* of the industrial case study

Case Study. In the case under study, of the 50 functional components identified, 20 contain source components belonging to a single layer each, while 8 contain source components outside layers. From the components per function visualization we were able to identify at least 3 large source components which attracted the attention of the system architects.

Components per Layer view

The *components per layer view* shows how the components are distributed across the application layers and provides an overview of the connections between the components. The presentation layer is at the top of the figure, while the business layer is at the bottom. Figure 4.5 simplify the identification of broadly used source components that implement complex functions with a significant number of lines of code. These components may require further investigation to verify if the logic they implement needs refactoring. Another interesting pattern is represented by wide hierarchies which have on the top a local component. In this case the local component is using and centralizing the logic from several global components. Developers might want to verify if the local extension needs to be split or the global components need to be merged.

Figure 4.5 shows the same source components of Figure 4.4 reorganized into layers. The only visualization difference lies in how the color gradients were computed, this time including all the components within the same layer.

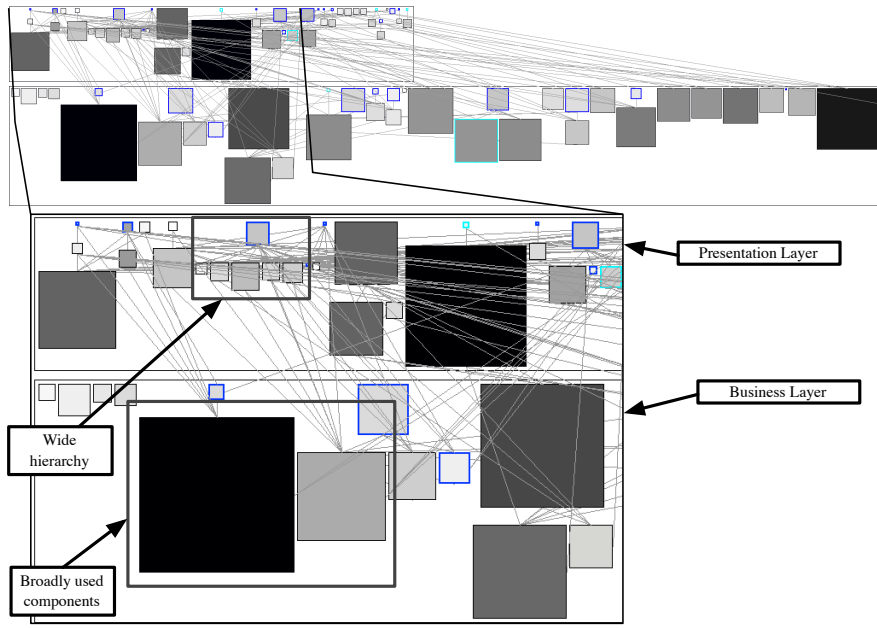


Figure 4.5: *Components per layer view* of the industrial case study

Case Study. In our case study we identified at least 6 components which contained a lot of lines of code and that were broadly used by other components. We also identified 3 hierarchies with 3 or more elements each that required further inspections.

4.3.2 Architectural Validation

The *components per layer view* and the *Components per function view* provide the user with an overview of the system from the architectural point of view. In this section we describe how we can exploit the same model used for architectural understanding to validate constraints on connections between architectural elements.

One of the main goal of our industrial partner was the identification of undesired associations amongst architectural components. We have been given the task to investigate the presence of four different types of undesired architectural associations:

1. Associations from the business layer to the presentation layer. Components belonging to the presentation layer are the entry points of the system functions, while components belonging to the business layer implement their business logic. Execution flows should always start from a function entry point and end with the code implementing the logic, not vice versa.
2. Associations from global components to local components. Global components must not be aware of the local components extending them.
3. Associations between global components. Global components should be deployed by their own so they need to be as independent as possible from other global components.
4. Associations between the business layers of different functions. The business logic of functional components must be all contained within their business layer.

The associations between architectural components are derived from source code dependencies. In particular, the architects of the banking system asked us to derive the architectural associations from method invocations and attribute accesses. These associations have been chosen because our architectural model is used to describe mainly elements of the Java source code. As a consequence, method invocations and attribute accesses are among the types of connections that best describe the nature of the associations we have been requested to identify.

After the deriving of component associations from the source code elements, we defined several queries on top of our architectural model to identify the undesired component dependencies. These queries validate the constraints on the dependencies between the components of our architectural model. We did not rely on any pre-existing framework or technique to query our model. Instead, the model queries have been defined using the model API. An example of a query performed on the model is shown in Listing 4.1. All the connections in the model are queried for associations between global components (lines 3–9 Listing 4.1). The purpose of this query is to verify the isolation between global components as requested by the architects of the banking system. The associations allowed by the `removeFalsePositives:` method in line 10 of Listing 4.1 are only: (1) associations from a newer version to an older version of the same component and, (2) associations from the presentation layer to the business layer of the same component.

```
1 | componentAssociations result |
2
3 componentAssociations := self model allComponentAssociations.
4
5 result := componentAssociations
```

```
6     select: [ :association |
7        association source isGlobal
8           and: [ association target isGlobal
9              and: [ association source ~= association target ] ] ].
10 result := self removeFalsePositives: result.
11
12 ^ result
```

Listing 4.1: Query to detect connections among global components.

Case Study. In the case study under analysis we found no associations from the business layer to the presentation layer and no associations from global components to local components. On the other hand we found 72 invalid connections among global components. 13 of them were identified by the architects as false positives. Also we found 12 connections between the business layers of different functions.

We implemented the undesired associations constraint required by the banking system architects in the reporting tool called Arki[7]. This tool makes it easy to inspect the constraint validation results and to apply the same constraint validation on different models. In Figure 4.6 we show how a typical Arki report looks like. The list of all the rules implemented is shown on the left panel, while the resulting model elements returned by a selected rule are shown in the interactive browser on the right side.

4.4 Conclusions

In this chapter we presented our approach to address two architectural problems in an industrial banking system. The first problem concerns the need of the system architects of an architectural overview to understand how the system components were distributed. To address this problem we designed and developed two polymetric views that provide an overview of the system from the points of view of the architectural layers and the functional components respectively. These visualizations can help the architects to identify problematical components in the system architecture. In particular, these visualizations help to identify: (i) complex local components that customize big portions of the application logic, (ii) local components that centralize the logic of several global components, (iii) components with several incoming or outgoing connections that cannot be deploy independently from other components.

7 http://www.themoosebook.org/book/internals/arki

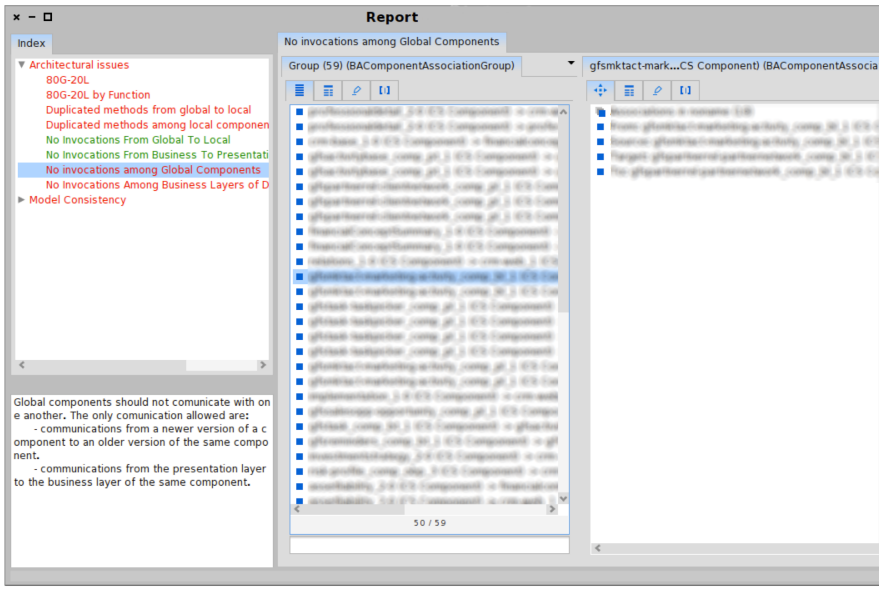


Figure 4.6: Arki reporting tool open on the connections between global components rule

The second problem concerns the validation of architectural constraints that the architects defined on the connections between the architectural elements of the banking system. To address this problem we formalized: (i) several queries to identify these constraint violations and, (ii) we provide the architects with a reporting system that integrates these queries.

The results of our analyses have been validated by the system architects who discovered problematic architectural elements they were not able to identify before.

The use cases presented in this chapter exploit our description of architectural elements and source code that are part of our approach to support the analysis of HAs. Because the architectural model and the model for source code languages are part of the same first-class description, it was straightforward to map architectural elements to source code ones. This homogenous model also was used to support the process of deriving architectural level dependencies from source code associations. The logic to derive dependencies, however, had to be implemented for all the elements of the architectural model which had implicit dependencies with other elements. This logic had to be replicated for each of the sources we considered (*i.e.*, method invocations and attribute access).

An architectural model should be capable of describing an architecture from the points of view of different stakeholders. Therefore, an architectural model should be able to represent the structure of an application as well as its domain and its business context. An architectural model should also contain elements at the right level of abstraction to support the analysis and the understanding of a system architecture. The component-connector representation which is largely used to define the structure of software architectures is too abstract to represent concrete architectural elements. In this dissertation we focused on the identification and reification of elements such as system layers which are the most common techniques used by software designers to decompose an application into parts. Other structural elements composing software architectures have to be identified and integrated into our description to make it more resilient to real architectures. Elements describing the business and the domain of an application should be identified and integrated in our model as well.

Chapter 5

Database Analysis

Relational databases are extensively used in the development of EAs because they provide an effective solution to the need of data persistence. A number of techniques and tools for software reverse engineering have been proposed in the last decade to support program comprehension, software maintenance and software evolution [Canfora *et al.*, 2011; Mens and Demeyer, 2008; Binkley, 2007; Tonella and Potrich, 2005]. Additional approaches and tools have been proposed as well for data reverse engineering with the aim of providing complete, up-to-date documentation of legacy data structures [Hainaut *et al.*, 2000; Mian and Hussain, 2008]. Few approaches, however, addressed the task of combined reverse engineering of both data and software. The combined analysis of databases and source codes of an application can lead to solving the challenges we face in the analysis of data-intensive systems [Cleve *et al.*, 2010b].

In this chapter we describe and validate a unified approach for the reverse engineering of both the databases and the source codes of EAs. The main goal of our approach is:

- To improve the understanding of the domain model of the application.
- To ease the integration of database systems with a more abstract representation of the data model.

Our approach uses the detection of object-relational persistence patterns to build a conceptual schema of an application. This conceptual schema is then used to map the domain model of the application thus providing the user with an overview of the overall system. We evaluate our approach on a large-scale open source enterprise system and discuss the obtained results.

This unified approach has been integrated in MooseEE and exploits our perspective on relational databases and source code (Figure 5.1).

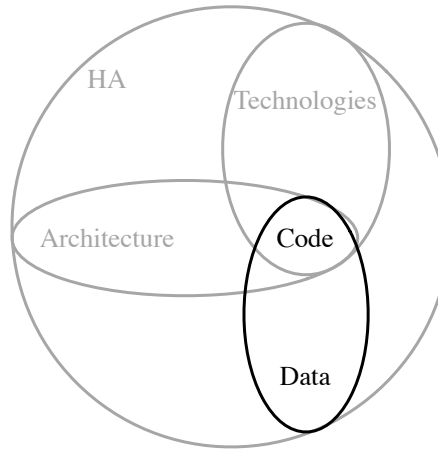


Figure 5.1: Point of view of data and code analyses

5.1 Related Work

Interest for data reverse engineering is increasing for businesses and organizations currently facing the critical issues of managing large amounts of data [Tallon, 2010]. The combined analysis of the database and the source code of an application can provide new ways to deal with the challenges we face to address so called data-intensive systems [Cleve *et al.*, 2010b]. Few approaches, however, have been proposed to address the reverse engineering of both data and software.

One of the most relevant techniques to deal with data intensive systems is the one proposed by Cleve *et al.* [Cleve *et al.*, 2006]. The authors defined a unified approach based on program slicing techniques applied to a system dependency graph in the presence of database statements. Another approach proposed by Cleve *et al.* [Cleve *et al.*, 2010a] exploits the automatic derivation of a relational database from a conceptual schema to support fast development of data-intensive systems. From the same schema is also generated a data manipulation API of the relational database.

Arcelli *et al.* [Arcelli Fontana *et al.*, 2010] proposed a data reverse engineering approach to generate a unified perspective on software and data by using design pattern detection. The approach comprises the creation of a conceptual schema using a unified perspective on an application software and data model.

In this work we leveraged the approach defined by Arcelli *et al.* and we provided full tool support for it. The implementation of this approach was made possible by our first-class description of the structure of relational databases and source code. Our

extensible meta-model for HAs was also extended to add a first-class description of the elements composing conceptual schemas.

5.2 Case Study

The case study used to validate our approach is ADempiere[1], an open source Enterprise Resource Planning (ERP) software. We can summarize the reasons why we identified ADempiere as an ideal candidate as follows:

Tiered architecture: The system manifests a clear separation between the different architectural tiers. To make data persistent the system uses relational database management systems (*i.e.*, PostgreSQL and Oracle).

Evolving and active system: The ADempiere project traces its evolution back more than a decade. Created in September 2006 as a fork of the Compiere open-source ERP, itself created in 1999, ADempiere soon reached the top ten of the *SourceForge.net* enterprise software rankings.

Large-scale and complex design: ADempiere is a multi-language system with a core written in Java that contains more than $3,805$ classes with more than half a million lines of code.

Model Classes	3,805
Methods	33,667
Lines of code	583,182
Tables	723
Table columns	13,892
Classes mapped on tables	1,402

Table 5.1: Metrics about the size of the case study

Table 5.1 lists six metrics related to the Java core and the database of ADempiere. The first three refer to the dimension of its source code. The last three refer to the size of the database and of the one to one connections between source code classes and database tables.

ADempiere has been designed in such a way that a developer can extend the system by modifying as little code as possible. Whenever a new table is added to the database, the Java code that represents that table at the source code level is automatically generated. Such a strict and well defined connection between the database and

1 http://www.adempiere.com

the application makes ADempiere a perfect workbench for the validation of our reverse engineering approach.

5.3 Software and data Reverse Engineering: a Unified Approach

Applications using relational databases to make data persist often exploit known architectural patterns, like, for example, the *Domain Model* and the *Data Mapper* [Fowler, 2005a], to represent persistent data at the source code level. The identification of these patterns can help in finding the links between the object model and other entities within an application. On the other hand, the conceptual schemas of a database have been proven to be an important resource in the reverse engineering process of large applications, in both the public and the private sectors [Batini *et al.*, 2011; Viscusi *et al.*, 2010; Batini *et al.*, 2006; Henrard *et al.*, 2007]. Our approach merges knowledge from pattern detection with the available information on the structure of relational databases to build the conceptual schema of an application. The elements composing the recovered conceptual schema are then mapped to the application software entities to provide a unified representation of the domain model of both the database and the application.

Our unified approach for data and software analysis follows several steps:

A1 *Extraction* of software and data information.

- *S1.1* *Extraction of software structure* to retrieve data on, *e.g.*, software classes, methods, generalizations and references.
- *S1.2* *Extraction of data entities* from logical/physical schemas.

A2 *Pattern detection* to identify uses of data entities (*e.g.*, DAO, persistence layer, *etc.*).

A3 *Concept reverse engineering* to identify and reify the concepts in the application.

- *S3.1* *Concept schema creation* by combining software and data analysis.
- *S3.2* *Mapping the concepts to the domain* to fill the gap between the data and the application.

Step *A1* is addressed by using the tools offered by the Moose ecosystem. In particular, to extract the software structure of ADempiere we used the Java parser *inFamix*[2]. To extract the database entities we used an SQL parser developed with *PetitParser* [Renggli *et al.*, 2010]. An importer implemented in MooseEE uses the SQL parser to populate our model for relational databases.

Step *A2* of our approach is addressed by the MARPLE design pattern detector [Arcelli Fontana and Zanoni, 2011]. The patterns we need to detect are the ones that can help identify the application connections to the database. For this reason we focused on the detection of the *Active Record* pattern [Fowler, 2005a]. MARPLE uses a rule based approach to detect the classes involved in a pattern implementation. The roles identified are:

- *Abstract Active Record*: a class representing the base implementation of a generic Active Record;
- *Concrete Active Record*: a class modeling a particular table;
- *Client*: every class using an active record which is not an Active Record or in the same hierarchy as an Active Record; the typical client is a business object composing the data provided by Active Records.

Active records in ADempiere are automatically generated during initialization by a procedure that accesses the information stored in the *Application Dictionary* (AD) section of the database of ADempiere. For each domain-related table specified in the *Application Dictionary*, a Java class named with the prefix "X_" and a Java Interface named with the prefix "*I_*" are generated by ADempiere. Each of these classes extends the *org.compiere.model.PO* class. MARPLE detected in ADempiere $1,324$ classes as implementors of an *Active Record* pattern. 995 classes, all extending the *org.compiere.model.PO* class, were identified as concrete active records. 91 classes, specializations of the *org.compiere.process.SvrProcess* class which provides server side processes with operations directly reading and writing on the database, were also identified as concrete active records. 13 classes were identified as client classes of some of the active records extending the *org.compiere.model.PO* class. The remaining 235 classes could not be related to database elements at source code level, nor as elements accessing the database. Therefore, they were tagged as false positives. Further research is needed to improve the accuracy of the MARPLE rule set, thus reducing the fraction of false positives.

2 http://www.intooitus.com/products/infamix/

5.3.1 Concepts Detection

Step *A3* of our approach (Section 5.3) involves the recovery of a conceptual schema of the system under analysis. To address this issue we provide a first-class description of the conceptual models that we included in our meta-model for HAs.

Figure 5.2 shows the subset of the extended meta-model for HAs where the new elements that model the conceptual schema are indicated in bold.

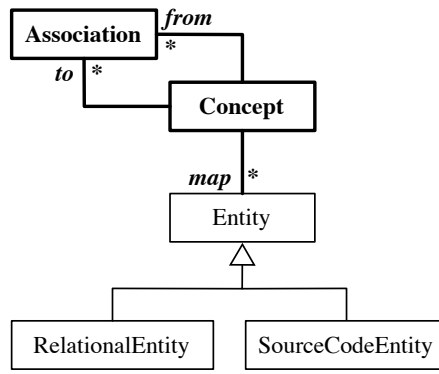


Figure 5.2: Conceptual schema meta-model

The meta-model for conceptual schemas includes the entities *Concept* and *Association*. The latter is used to put a concept in a relationship with other concepts. The entity *Concept* can map to other entities representing source code or relational elements.

We populate our model for conceptual schemas by using the information obtained in the steps *A1* and *A2* of our unified approach. In particular, the algorithm we use to identify concepts (step *S3.1*) takes as input the collection of classes identified in the step *A2* and can be described as follows:

1. The names of the input classes are analyzed and a list of all the possible pairs is generated.
2. The longest common substring algorithm is applied to each pair.
3. For each common substring identified, a set containing all the classes that include that substring in their name is created.
4. The relational elements connected to each set of classes are added to the set itself.

5. Finally, for each set a concept is instantiated that maps to the entities within that set.

The output of the algorithm is a collection of concepts. Each of these concepts is named as the longest common substring computed amongst the elements it maps to. From the results we filtered out all the concepts with names at least 55% shorter than the length of the class names they mapped to. This avoided the creation of concepts with meaningless names and that map to classes sharing no rationals. The empirical threshold of 55% was determined by trial-and-error and it is likely to be highly system-dependent.

The application of our algorithm to ADempiere resulted in a total of 277 identified concepts. In Tables 5.2 and 5.3 we list the top 10 and the last 10 identified concepts, respectively, ordered by name length. All the concepts identified have meaningful names or have a name that is a concatenation of meaningful words with few exceptions, *e.g.*, in line 2 Table 5.2 the concept name is contaminated by the prefix "*P_*". Only 2 of the 277 concepts identified have a name with no meaning: *eference* and *ocation*. The concept *eference* mapped to the classes *X_AD_Reference*, *MPreference*, *WPreference* and the table *ad_reference*. These classes should have been mapped by the concepts *Reference* and *Preference* respectively and not by the concept *eference*. The concept *ocation* mapped to the classes *WAllocation*, *X_C_Location*, *MLocation* and the table *c_location*. Also in this case these elements should have been mapped by the concepts *Allocation* and *Location* respectively. *Preference*, *Allocation* and *Location* are existing concepts while *Reference* is not. The reason lies in that the name "Reference" matches only the active record *X_AD_Reference*. 637 classes identified as active records have not been included in the concept list by our algorithm for the same reason. This observation suggests that the active records that have not been added in any concept can become a concept on their own if they are at least related to a database table.

The concept identification procedure described in this section has to be understood as a semi-automatic way to infer the concepts implemented by an application and to detect the elements contained in the identified concepts. The result of the concept inference should be manually checked by the user to verify its accuracy.

5.3.2 Mapping the domain model to the conceptual schema

The concepts identified in step *A3* of our approach represent an initial schema to support the understanding of an application and its data structures. With step *S3.2*, however, we want to enrich the concepts identified with the elements composing the domain model of the application to provide a complete picture of the system.

	Concept	Contained Entities
1	DistributionRunDetail	model::MDistributionRunDetail, model::X_T_DistributionRunDetail, t_distributionrundetail
2	P_ProcessorParameter	model::X_EXP_ProcessorParameter, model::X_IMP_ProcessorParameter, exp_processorparameter, imp_processorparameter
3	AttributeSetInstance	model::X_M_AttributeSetInstance, m_attributesetinstance, model::MAttributeSetInstance
4	BankStatementMatcher	model::X_C_BankStatementMatcher, c_bankstatementmatcher, process::BankStatementMatcher
5	LandedCostAllocation	c_landedcostallocation, model::MLandedCostAllocation, model::X_C_LandedCostAllocation
6	WorkflowProcessorLog	model::X_AD_WorkflowProcessorLog, wf::MWorkflowProcessorLog, ad_workflowprocessorlog
7	NetworkDistribution	dd_networkdistribution, dd_networkdistributionline, org::eevolution::model::X_DD_NetworkDistributionLine, org::eevolution::model::X_DD_NetworkDistribution
8	BankStatementLoader	model::X_C_BankStatementLoader, c_bankstatementloader, model::MBankStatementLoader
9	RequestProcessorLog	r_requestprocessorlog, model::MRequestProcessorLog, model::X_R_RequestProcessorLog
10	DistributionRunLine	model::MDistributionRunLine, model::X_M_DistributionRunLine, m_distributionrunline

Table 5.2: First 10 concepts ordered by name length

The domain was defined by Fowler [Fowler, 2005a] as: *An object model of the domain that incorporates both behavior and data*.

According to this definition, to reconstruct the domain model of an application, we need to identify both the elements mapping the data and the elements implementing the application logic. The concepts identified in step *A3* of our approach already map to the elements of the application and the database that represent the data. Now we need to identify the elements that implement the behavior of the application and include them in the our conceptual model as well.

The following formula defines how we identify the domain model elements contained in a concept:

$$\beta^*(\omega(\alpha^*(M_c))) \qquad \forall c \in C \tag{5.1}$$

where:

$$
\begin{aligned}
C &= \text{all concepts} && (5.2)\\
CLS &= \text{all classes} && (5.3)\\
M &= \text{all methods} && (5.4)\\
I &\subseteq M \times M \text{ are invocations} && (5.5)\\
INH &\subseteq CLS \times CLS \text{ are inheritances} && (5.6)\\
Meth &\subseteq M \times CLS && (5.7)\\
\gamma &: CLS \rightarrow 2^C && (5.8)\\
\alpha(y) &= \{x|(x,y) \in I\} && (5.9)\\
\beta(cls) &= \{cls'|(cls,cls') \in INH\} && (5.10)\\
\omega(m) &= \{cls|(m,cls) \in Meth\} && (5.11)\\
M_c &= \{m|m \in M \wedge c \subset \gamma(\omega(m))\} && (5.12)
\end{aligned}
$$

C is the set containing all concepts in the conceptual schema. I represents the invocation relation between methods. $(x, y) \in I$ means x calls y. INH represents the inheritance relation between classes. $(cls, cls') \in INH$ means cls subclasses cls'. $Meth$ represents the relation between methods and the classes defining them. $(m, cls) \in Meth$ means m is defined in cls. $\gamma(cls)$ returns the concept $c \in C$ which contains the class $cls \in CLS$. $\alpha(y)^*$ returns all methods x *preceding* y in some invocation chain. $\beta(cls)$ returns the set of subclasses cls' of cls. $\omega(m)$ returns the class $cls \in CLS$ which defines the method $m \in M$. Finally, M_c is the set of methods belong to the concept $c \in C$.

We defined an algorithm to compute Definition 5.1 that follows 3 steps: (1) we evaluate $\alpha(M_c)$ to retrieve the static invocation chain for each class method contained in a concept, (2) we apply the ω function (Definition 5.11) to retrieve the set of classes calling the methods in M_c (Definition 5.12), (3) we compute the transitive closure of β (Definition 5.10) applied on the classes retrieved in the previous step to find all their subclasses.

This algorithm returns for each concept all the classes that implement the application logic of the concept. The results obtained by applying this algorithm on ADempiere were included in our conceptual model. Consequently, the concepts in our model map also to the elements implementing the application logic. After the integration we observed that the number of elements mapped by some concepts increased conspicuously. Some concepts included in excess of 400 elements. Such large numbers are due to the depth of the inheritance hierarchies of the classes mapped by the relevant concepts. Concepts that map to several elements of the

	Concept	Contained Entities
268	Report	model::X_T_Report, www::WReport, print::AReport, t_report, report::FinReport
269	Order	org::eevolution::model::MDDOrder, model::MOrder, c_order, process::OrderOpen, model::X_C_Order, acct::Doc_Order, i_order, model::X_I_Order, model::MOrderTax, wstore::WebOrder, process::CopyOrder
270	Index	k_index, model::MIndex, model::X_K_Index
271	Store	w_store, model::X_W_Store, model::MStore
272	InOut	model::X_M_InOut, acct::Doc_InOut, model::MInOut, m_inout
273	Topic	model::X_K_Topic, b_topic, model::X_B_Topic, model::MRfQTopic, k_topic
274	Asset	i_asset, a_asset, model::MAssetUse, model::X_I_Asset, model::MAsset, model::X_A_Asset
275	Click	wstore::Click, model::X_W_Click, w_click
276	Group	model::X_R_Group, model::MBPGroup, r_group
277	Aging	model::X_T_Aging, model::MAging, process::Aging, t_aging

Table 5.3: Last 10 concepts ordered by name length

domain model should be considered more important than others since their implementation covers a larger portion of the application.

Our algorithm to implement Definition 5.1 is based on source code information. The accuracy of this algorithm could be increased by using dynamic information to compute Function 5.11. This will reduce the number of elements mapped by the concepts, therefore providing a more specific conceptual model of the application under analysis. Another option to improve the quality of the results would be to avoid the computation of Function 5.10. This option could be considered as a good approximation if we would keep using static information to compute Function 5.11. We should also consider that the unified conceptual schema we propose is meant to provide an overview of an application together with its data model. Therefore, more complete or specific information about the elements composing a concept could then be computed lazily during a manual inspection.

5.3.3 Connecting concepts

After creating a conceptual model that includes elements from the data layer, the database and the domain model, we need to define the rationales of the connections between the concepts of our model.

We decided to connect the concepts by considering their shared elements. Two concepts are defined as connected if they map to one or more shared element. For example the concept "BankAccount" and the concept "Invoice" are connected because they both map on several common elements like *e.g.*, the class "PaymentServlet". This class has been evidently added because part of the invocation chain of both the concepts "BankAccount" and "Invoice". This kind of association between concepts takes into account both database tables and source code entities. Other kinds of connections are of course possible: for example, connections derived from the foreign keys between tables or connections derived by source code elements like method invocations.

By connecting the 277 concepts of the conceptual model of ADempiere we identified $21,162$ connections derived by shared elements. 81 of the these connections have been derived by shared database tables. This means we have an average of 76.4 connections per concept. Such a large number of connections is due to the consistent number of classes mapped by some concepts after we added the classes belonging the domain model of ADempiere.

5.4 Conclusions

In this chapter we presented an approach to support reverse engineering and evolution of data-intensive systems. This approach tries to answer to the challenges pointed out in the literature [Cleve *et al.*, 2010b] by providing reverse engineers with a unified representation of data-intensive systems for the understanding of both software and data semantics.

Threats to *external validity* are concerned with generalization of our findings. Although we performed our evaluation on a large-scale enterprise system, which is representative of the state of the art of enterprise systems developed in Java, we are aware that more studies are required to be able to generalize our findings. On the other hand the instruments used to implement the approach in Section 5.3 are generic and can be used without modifications on other JEAs. Threats to *construct validity* are concerned with the quality of the data we have analyzed, and the degree of manual analysis that was involved. Our approach is meant to be a semi-automatic support tool to reverse engineer heterogeneous systems that rely on re-

lational databases. Therefore, a manual validation and refinement of the results is required. The workload of the manual inspection, however, can vary from system to system accordingly the complexity of the application analyzed and the accuracy required by the user.

The approach presented in this chapter is based on our first-class description of relational databases and source code languages. Our homogenous model for HAs enables the creation of a conceptual schema of the application in the first place. By integrating the conceptual schema of the application in our model for HAs we were able to analyze the dependencies between the concepts of our application elements. Our conceptual recovery approach requires further development to improve its accuracy, however, we demonstrate the effectiveness of our unified model to support the analysis of HAs.

In this dissertation we focused on describing the structure of relational databases. To extend our analysis on the actual data stored within a database we would likely need a different first-class representation from the one we propose.

Relational databases are one way to make data persistent. First-class descriptions of other persistent data storages would be required to analyze a broader spectrum of HAs.

Chapter 6

Transaction Flow

In this chapter we address the problem of the identification of transaction scopes in JEAs by exploiting our first-class description of JEA specific technologies and source code (Figure 6.1). We show how we can address the different facets of this problem by exploiting also the other perspective on architectures and relational databases included in our approach to analyze HASs.

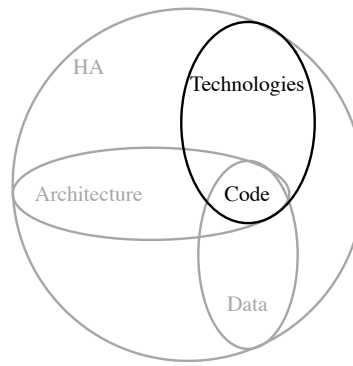


Figure 6.1: Technological perspective on HAs

Application transactions have been defined in JEAs to guarantee atomicity of operations and to provide a certain level of isolation of services accessed by different clients. Transactions aid the application developers in achieving the desired level of reliability and consistency. On the one hand, it is important to ensure that critical services are properly contained within a transaction scope, while, on the other hand, starting unnecessary transactions should be avoided for performance reasons. We define a transaction as unnecessary if its scope is nested within another transaction scope.

In Java 2 Enterprise Edition (J2EE), a method can be defined as part of a transaction by setting certain attributes in the XML deployment descriptor file of the applica-

tion. This file specifies which are the system classes implementing an Enterprise Java Bean (EJB) [DeMichiel, 2003] and which is the transaction attribute associated with the class methods. The EJB container takes these attributes into account to manage application's transactions. J2EE specifications defines various transaction propagation semantics. The most common attribute is *required* which creates a new transaction if none is active, or reuses the existing one. A method without an explicit transaction attribute can still be part of a transaction if it is has been invoked by another transactional method.

The advantage of this declarative definition is that it is not necessary to touch the Java source code to manipulate them. Nevertheless, the decoupling between the actual method and its transaction attribute makes it difficult to identify the transaction scope the method lies in. Consequently, developers cannot easily verify if a method is transactional or not if they need to encapsulate operations within a transaction scope. An example of an operation that needs to be contained within a transaction scope is the lazy loading with ORM (Object-Relational Mapping). This kind of operations cannot be executed if a transaction has been already committed. Furthermore, a transaction can either be started manually using the Java Transaction API or it can be started automatically in applications by Container-Managed Transactions. As a consequence, the usage of transactions is not uniform, further complicating their understanding.

Our analysis of the application transaction scope looks for an answer to four specific questions:

1. Which methods are involved in a transaction scope?

2. Which methods start unnecessary transactions?

3. How are the methods involved in a transaction scope distributed in the application architecture?

4. Which methods access the database from outside a transaction?

The first two questions can be addressed using our first-class description of EJBs by merging information from the application source code and the XML deployment descriptors. To address the third and fourth questions we need to look at the problem of identifying application transactions from the point of view of the architecture and data persistency respectively.

In this chapter we present a technique to identify methods and classes involved in application transactions and we propose three visualizations that aim to highlight anomalies in their definitions.

We applied our approach to two different versions of an industrial case study consisting of over 1500 Java classes. The technique to identify the transaction scope has been validated by manual inspection mainly using Eclipse. The manual inspection confirmed a good of precision in the identification of all EJBs and of all other elements shown in the visualizations. These results have been presented to the company that provided us with the case study, and their feedback was highly positive.

6.1 Transaction Flow Identification

Leveraging our meta-model for EJBs described in Section 3.3 we can tackle the problem of identifying transaction scope in JEAs. Application transactions in Java guarantee isolation among services and group multiple operations in a unique unit of work. It is not trivial to identify methods involved in a transaction because this is a property with both dynamic and static aspects. A method may be transactional either because it is specified in the deployment descriptor or because it overrides or is invoked by a method that is part of a transaction. We consider the following methods to be part of a transaction [Armstrong *et al.*, 2005]:

1. Methods that start a transaction (*i.e.*, their transaction attribute is 'Required' or 'RequiresNew').

2. Methods that override methods that start a transaction.

3. Methods with a transaction attribute 'Mandatory', 'Required' or 'Supports' that are invoked by methods already being part of a transaction.

4. Methods without a transaction attribute that are invoked by methods already part of a transaction.

To expose the transaction scope of methods we proceed in the following way: We define a set of all methods that have a transaction attribute defined. So the initial set includes methods that start a transaction, those that override a method that starts a transaction, and methods that support transactions (a method *supports* a transaction if its transaction attribute is not defined or it is different from 'never' and 'notSupport' [Armstrong *et al.*, 2005]). Then we traverse breadth-first the invocation tree of methods invoked by this set. Using this technique we are able to identify all methods that belong to a transaction scope or that support transactions triggered from outside the application, *i.e.*, from the UI. Before explaining how we identify the invocation paths that are encapsulated in a transaction scope we need some definitions:

Entry point method: is a method that is not invoked by other methods.

Safe path: is an invocation chain that starts from an entry point method involved in a transaction.

Unsafe path: is an invocation chain that starts from an entry point that does not start a transaction.

The following formula describes how we identify an unsafe path:

$$\alpha(Q) \quad \cap \quad \omega(\alpha(Q) \cap E) \tag{6.1}$$

where:

$$\begin{aligned} M &= \text{all methods} && (6.2)\\ I &\subseteq M \times M \text{ are invocations} && (6.3)\\ E &\subseteq M \text{ are entry points} && (6.4)\\ T &\subseteq M \text{ start a transaction} && (6.5)\\ Q &\subseteq M \text{ perform a query} && (6.6)\\ \omega(x) &= \{y|(x,y) \in I^*\} && (6.7)\\ \alpha(y) &= \{x|(x,y) \in I^* \text{ and } x \notin T\} && (6.8) \end{aligned}$$

M is the set containing all methods of the system. I represents the invocation relation between methods. $(x, y) \in I$ means x calls y. E, T and Q are respectively the subset of methods that are entry points, the subset of methods that start a transaction and the subset of methods that perform a query. I^* is the transitive closure. Finally, $\alpha(y)$ returns all methods x *preceding* y in some invocation chain, and $\omega(x)$ returns all methods y *following* x in some invocation chain.

Figure 6.2 illustrates the steps of an algorithm to compute Equation 6.1 and compute *unsafe paths*. Step 1 consists in the evaluation of $\alpha(Q)$. In accordance with its definition, the result of $\alpha(Q)$ is the set of methods that are part of the invocation chain ending with a method that executes a query and starting with a method that does not start a transaction. Moreover, none of the methods returned by this function start a transaction. Step 2 is to apply ω to the intersection $\alpha(Q) \cap E$. The result of $\omega(\alpha(Q) \cap E)$ is the subset of methods contained in $\alpha(Q) \cap E$ that are part of the invocation chain starting from an entry point that does not start a transaction. Step 3 consists in the evaluation of the intersection of $\alpha(Q)$ and $\omega(\alpha(Q) \cap E)$ in order to clean up the set from methods not strictly related with the unsafe path.

The result is a set containing the methods of all unsafe paths, namely invocation paths starting from an entry point that do not start a transaction and ending with a method accessing the database. In the diagram, the black path is unsafe since it ultimately performs a query without ever starting a transaction.

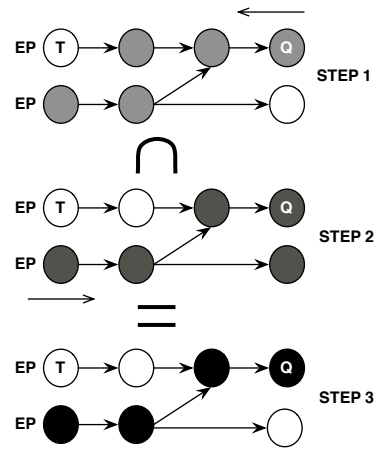


Figure 6.2: Unsafe path identification

6.2 Visualizations

To inspect a JEA from the point of view of application transactions, we devise three interactive visualizations:

- *Transaction flow* provides an overview of the methods involved in transactions (Figure 6.3),
- *Server Layers* offers an overview of the typical layers in a JEA and helps to identify the misplaced transaction specifications (Figure 6.4), and
- *Unsafe queries* reveals the methods that query the database without being covered by a transaction (Figure 6.5).

These visualizations have been developed using the Mondrian visualization engine [Meyer *et al.*, 2006].

In this section we explain in detail each of the three visualizations. We furthermore evaluate the effectiveness of each visualization to analyze an industrial content management system (CMS) to manage customer data. We have analyzed two versions of the same case study released a year apart. The older version is composed of 1938 classes and 55 EJBs, while the newer version is composed of 1527 classes and 43 EJBs. Figures 6.3, 6.4 and 6.5 show elements of the newer version of the case study.

6.2.1 Transaction flow

Figure 6.3 shows all classes and their methods involved in a transaction according to the criteria explained in Section 6.1. Classes and methods are presented as hierarchies that express invocation order. This means that methods of classes on top invoke methods of classes below them. Invocations among classes are represented by grey edges. Internal invocations among methods of the same class instead are organized as a hierarchy going from left to right. The colors of the elements in Figure 6.3 have the following meaning:

1. *Blue methods* start a transaction. (1) and (7) in Figure 6.3.

2. *Cyan methods* have a transaction attribute equal to 'Mandatory', 'Required' or 'Supports' and are invoked by methods involved in a transaction. (2) in Figure 6.3.

3. *Magenta methods* have a transaction attribute equal to 'RequiresNew' and are invoked by methods that start a transaction. (5) in Figure 6.3.

4. *Grey methods* have no transaction attribute and are invoked by methods already part of a transaction. (3) and (4) in Figure 6.3.

5. *Orange methods* are entry point methods that have a transaction attribute equal to 'Supports'. (6) in Figure 6.3.

The Transaction Flow visualization makes it easy to identify all methods that start an unnecessary transaction (magenta methods, (5) in Figure 6.3). These methods in fact start a transaction when they could simply use the transaction scope of their invoker method. We say that they start an *unnecessary transaction*. Such methods have to be manually checked to verify whether the nested transaction is useless or not. For example a new transaction can be explicitly required to log a message in a database independently of whether the main transaction commits or rolls back.

All hierarchies having no methods starting a transaction at the beginning (blue methods) can lead to problems. These methods support transactions but they do not start one by themselves. This means that either they are within the scope of a transaction started from the application front-end (generally a web interface built using JS Pages but also possibly a GUI written in Java), or the service it uses lies outside a transaction scope. We call such hierarchies *weak paths*. The Transaction Flow visualization is also useful to identify isolated parts of the code that are independent of the rest of the application. In Figure 6.3 we see two isolated hierarchies at the top left. Hierarchies like these are interesting to identify because they represent services that are "self-contained" in the sense that their entry point and their

implementation is not related to other elements of the application. Such hierarchies may also be a sign that opportunities for sharing logic between services have not yet been exploited. On the other hand it is also possible to identify more complex hierarchies with multiple entry points sharing various classes. The identification of these structures may be useful to guide refactoring to make application services more independent. We now evaluate the effectiveness of these visualization on the latest version of an industrial case study.

Case Study. In the case study under analysis all Session beans have methods with a transaction attribute defined and almost all methods start a transaction. We can count 489 methods starting a transaction and 1537 methods involved in a transaction scope. All Session beans appear in the top of the invocation chain, which means that they are actually used as access points for the application services. Using the visualization from Figure 6.3 we can identify methods that unnecessarily start a transaction. In our case we detected 5 cases.

At the right and in the middle of Figure 6.3 there are two classes containing methods that support transactions but do not start one. The invocation chains starting from those methods are the only weak paths in the case study. In this case it is necessary to check if the transaction is started in the front-end. If this is the case the EJB container will propagate the transaction scope from the front-end, otherwise these methods are actually operating on the system

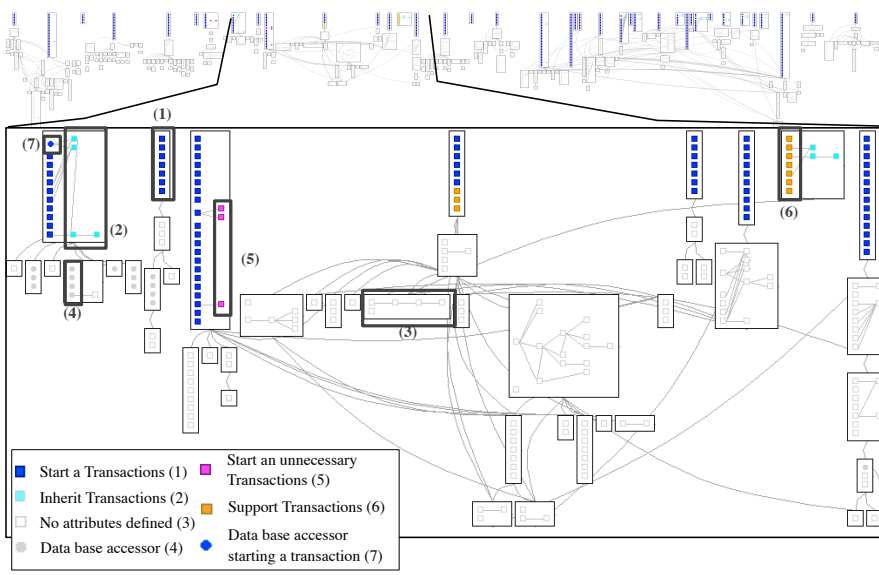


Figure 6.3: An excerpt from a Transaction Flow Visualization

outside a transaction scope. By showing the entry points that are a potential issue, we ease the investigation of the front-end.

At the left part of Figure 6.3 there are two small hierarchies starting from a Session Bean without external incoming or outgoing edges. This suggests either that the services that they implement are logically independent from other services, or that the potential for sharing logical functionality with other services has not been exploited. We say that such hierarchies have a "service-oriented" design. In Figure 6.3 there are also some more complex hierarchies with multiple entry points sharing various classes. In contrast to the "service-oriented" hierarchies at the far left, these more complex hierarchies have multiple entry points that share behavior. We say that such hierarchies have a "use case-oriented" design, since the implementation classes support multiple services.

A final point is the method at the top-left of Figure 6.3. This method is blue, so it starts a transaction, and it is round, so it accesses the database. This means that the Session Bean it belongs to contains behaviour that is not supposed to be implemented at this level. Ideally the database access should be contained in dedicated classes and not directly in Session Beans. This discovery motivates another visualization, explained in Section 6.2.2, which is related to the Transaction Flow but has the purpose of identifying architectural violations.

6.2.2 Server Layers

Session Beans that start a transaction are used as entry points for services. It is considered good practice to split EA components into presentation, domain and data layers [Fowler, 2005a]. The domain layer can be further split into a service layer and a domain model. Figure 6.4 shows the same classes of Figure 6.3 reorganized into layers. The layer on top is the service layer, in the middle there is the logic layer, and on the bottom there is the data layer. The criteria to arrange classes into layers are as follows:

1. In the Service layer are visualized all classes implementing a Session bean.

2. In the Data layer we show all classes that (1) are part of the invocation chain starting from the classes belonging to the service layer, and that (2) execute a query, implement the Java interface *Serializable*, or throw an exception from the *java.sql* package. In the case the class implements the interface *Serializable* it can be useful, in order to ensure that the class is actually used to communicate data, to check if the class is also a *data class* [Lanza and Marinescu, 2006].

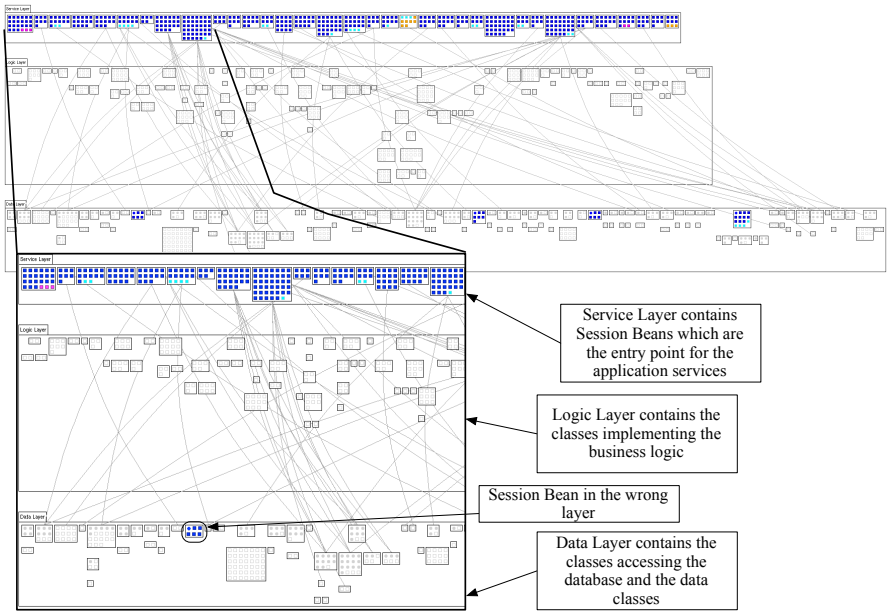


Figure 6.4: An excerpt from a Server Layers Visualization

3. In the Logic layer we show all classes that are part of the invocation chain that started from classes belonging to the service layer, but that are not part of the data layer.

Colors and shapes of objects in Figure 6.4 have the same meaning as in the Transaction Flow visualization. Edges however have a different meaning: the grey edges are invocations that jump a layer by going from the Service layer directly to the Data layer. The purpose of this visualization is to expose structural violations of the elements involved in a transaction. In particular we highlight all Session beans that are in the wrong layer and all invocations that jump a layer. Such violations of architectural constraints not only impact program comprehension, but they may be signs of more serious defects in the application code. The layering strategy we used in our approach can of course fail to match a custom layering structure. Our approach, however, is flexible and extensible so to ease the definition of layering strategies suitable for specific architectures.

Case Study. In Figure 6.4 there are several edges from the session layer directly to the data layer, suggesting that many services apparently do not have any business logic. Either the business logic has been moved into the Session bean or into the classes belonging to the data layer. In both cases we are dealing with classes that implement behavior external to their competence. We can count

16 Session Beans with grey edges starting from them. Another possibility is that a particular service has no business logic so there are no classes belonging to that layer. This case is also problematic because a service should normally touch all layers — a service without business logic is suspect because it directly exposes the data layer. In any case, software comprehension is compromised because knowledge about the purpose and behavior of a service should appear in the logic layer.

Figure 6.4 also shows a Session Bean positioned in the wrong layer (in the whole system we identify four of them). These Beans have been classified as belonging to the data layer because they have at least one method that accesses the database. There are two possible interpretations why this is so: either the service is used to send some data back to the front-end without performing any computation, or the business logic has been pushed to the Session bean instead of creating a separate class belonging to the logic layer. This is similar to the problem related to invocations jumping a layer. In this case we have discovered methods with a behavior outside of their competence.

6.2.3 Unsafe Queries

If on the one hand we are able to identify methods involved in a transaction, on the other hand we can show which methods perform queries outside of a transaction. Identifying methods that access the database outside a transaction scope is important to avoid problems of consistency. These methods cannot know if they are working with consistent data or not. Instead, participating in a transaction ensures that isolation is respected even in the presence of multiple concurrent users of the system. The Unsafe Query visualization shows those hierarchies that end with classes that execute a query to the database and are outside a transaction scope. In particular:

- *black classes* are test classes,
- *grey classes* are not test classes,
- *yellow methods* perform a select,
- *orange methods* perform an update, and
- *red methods* perform both.

The organization of classes is the same as in the Transaction Flow: classes are shown considering the invocation order from the top to the bottom. Edges represent the

invocations between methods. Considering this organization, in the top part of the Unsafe Queries visualization are the entry points for unsafe paths. At the bottom, the visualization shows classes that actually perform queries on the database.

Test classes are identified using a naming convention: A class whose name matches the regular expression *".*Test.*"* or a class contained in a package hierarchy whose name matches the regular expression *".*test.*"* is considered to be a test class.

Case Study. Considering our case study which is partially shown in Figure 6.5, we can see that almost all hierarchies start with a test class. This means that during normal execution these paths have to be considered safe. We can count 562 methods outside a transaction scope. In the left part of Figure 6.5 there is a hierarchy that starts with a grey class. This is an unsafe path that actually accesses the database by reading data without being sure that they are consistent. We count 24 methods belonging to this hierarchy. Also interesting are the two grey classes on the far left of Figure 6.5. These classes contain a method that performs an update of the database (they might contain other methods that are omitted in the visualizations) and they are not invoked by other methods. These methods pose a risk since they may be invoked directly from the front-end without being part of a transaction scope.

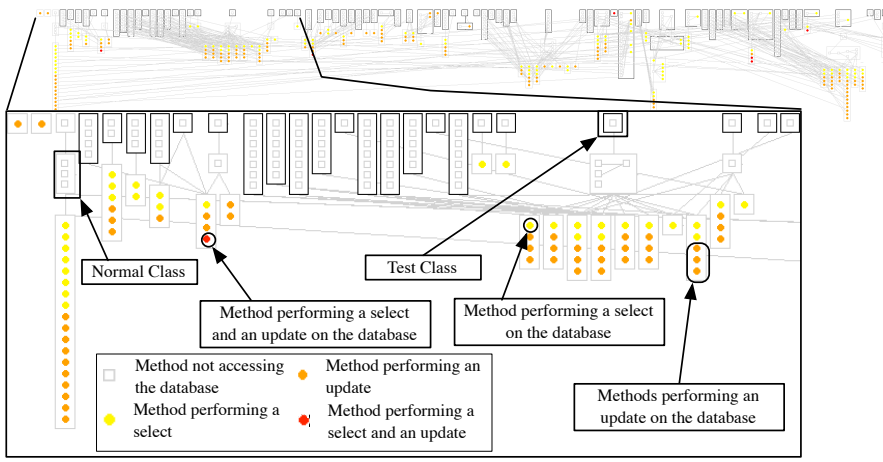


Figure 6.5: An excerpt from an Unsafe Queries Visualization

6.3 Comparison of Case studies and Validation

In this section we compare the two versions of our case study using the visualizations presented in the previous section to assess the overall quality of the applica-

tion. In Table 6.1 we present some selected metrics related to the systems under analysis to give an idea of the dimension and complexity of the case studies.

	Old	New
Classes	1,938	1,527
Session beans	49	39
Message-driven beans	6	4
Entity beans	0	0
Averave number of methods per bean	14.29	13.15
Methods starting unnecessary transaction	13	5
Methods starting a transaction	622	489
Methods involved in a transaction	2,315	1,537
Methods involved in an unsafe path	589	562
Isolated call-hierarchies	10	15

Table 6.1: Selected case study metrics

Transaction Flow: In Figure 6.6 we show the Transaction Flow visualizations of both versions. The visualization shows that the application has been strongly modified. The number of single hierarchies without external relations has increased even if there is still a huge hierarchy with many entry points and a large number of classes implementing business logic. In the previous version of the code 10 hierarchies were identifiable, in the new version 15. This difference suggests that in the past the application has been designed and developed considering the point of view of use cases. In the refactoring process many classes have been eliminated and the application adopts a design that is much more service oriented.

Visually, it is also possible to notice that hierarchies are not only cohesive, but also deeper. More classes and methods were involved in the business logic needed to fulfill a service requests. The number of methods starting an unnecessary transaction are reduced from 13 to 5 and the number of beans go down from 49 to 39 which means that the implementation of the services has been modified to eliminate unnecessary transactions. To summarize, all modifications that have been performed on the application improve its structure. The new version of the application is much closer to what we expect from the structure of a JEA: small call-hierarchies with Session beans on top, without relations to other hierarchies and without methods starting unnecessary transactions. Instead all the services are self-contained and transactions are always started if necessary.

Server Layers: In Figure 6.7 are shown the Service Layers visualizations of both the system versions. In the new version of the system we identified that the logic layer is thinner. This means that the invocation hierarchies are less deep and so that either the logic or the interaction between classes have been simplified in the new system.

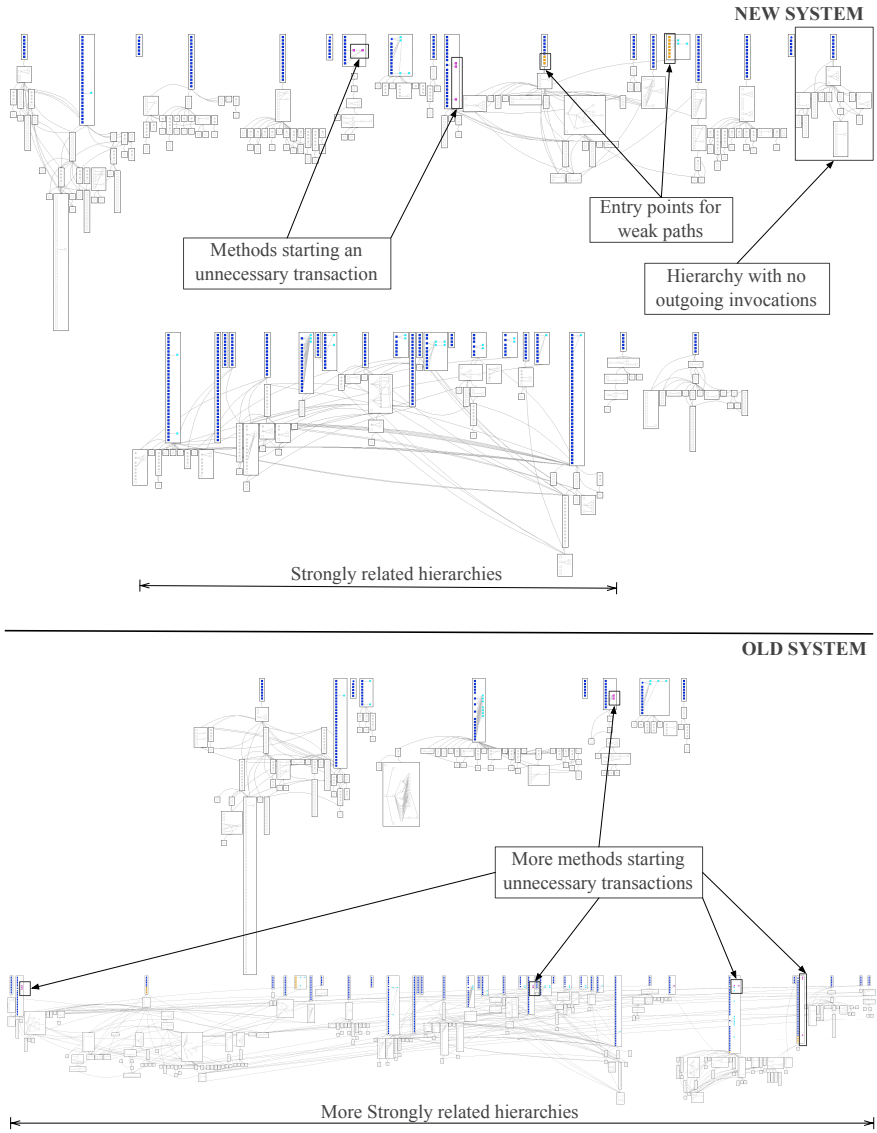


Figure 6.6: Transaction Flow visualization of two versions of the same case study

In the Transaction Flow visualization was visible that some hierarchies have been refactored, however, only in the Service Layer view we could identify where the developers focused their attention during the refactoring. The layered organization of the classes highlights that during the refactoring process many classes involved in the computation of requests from services have been removed. Also there are fewer edges so the communications between the system classes have been changed. This is maybe because of the lower number of Session Beans but also due to a better organization of the responsibility inside the application.

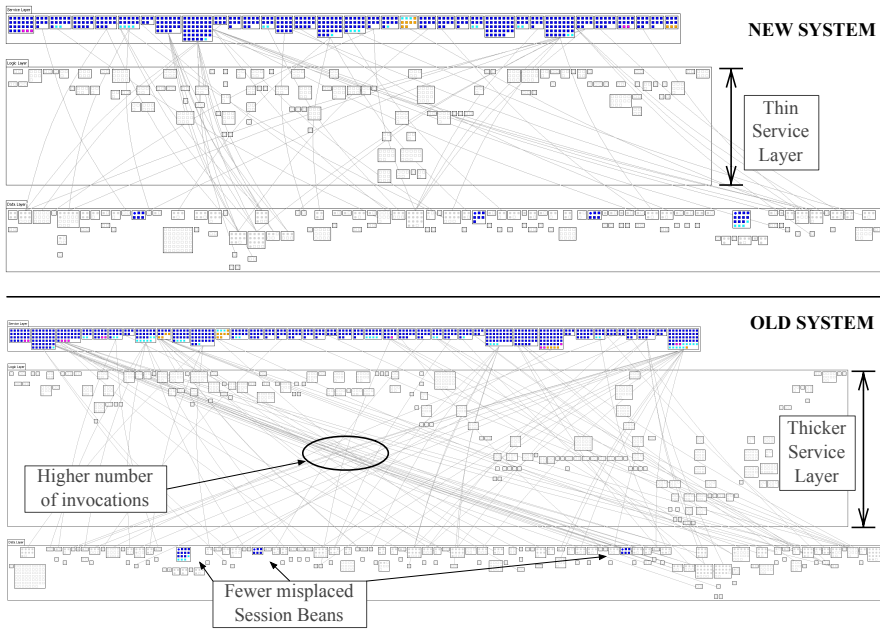


Figure 6.7: Server Layer visualization of two versions of the same case study

Unsafe Queries: Figure 6.8 depicts the Unsafe Queries visualizations of the old and the new versions of the system. In the old system version we can identify 5 lonely classes that access the database inserted in a hierarchy. In total 10 unsafe paths were identified against 3 in the new version. Also in this visualization it can be seen how the refactoring process not only changes the structure but also makes the application's services safer and more efficient.

The case studies were helpful to validate the tool and verify that our technique works correctly not only on small sample code. The validation has been performed manually mainly using Eclipse. First, we verified that all beans contained in the deployment description have been imported into the model. In both versions we correctly imported all beans. Second, we also manually verified that the transaction

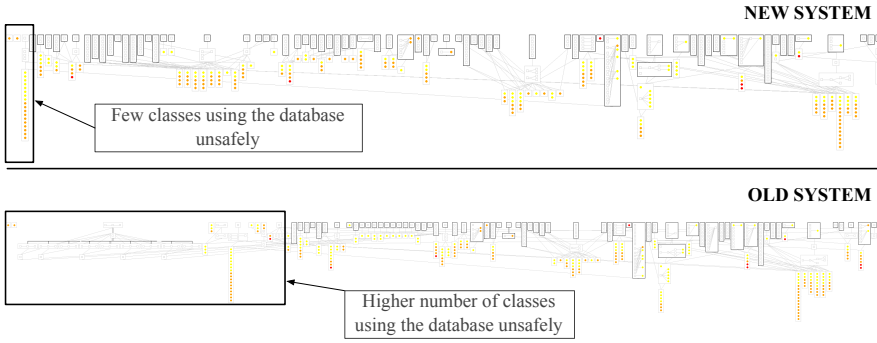


Figure 6.8: Unsafe Queries visualization of two versions of the same case study

attribute is correctly defined for the beans. In this case all methods have been assigned the right transaction attributes. Third, using the Call Hierarchy function of Eclipse we investigated manually a set of random invocation paths in the real case study confirming their correctness.

Considering the Server Layers visualization, all Beans were present in the service layer besides the 4 that access the database. In the Data layer we were able to identify all elements accessing the database using the java.sql package.

Considering the Unsafe Queries visualizations we detected that the methods considered to be entry points for unsafe paths were actually not invoked by other methods.

The manual inspection confirmed a high level of precision in the identification of the elements composing the visualizations. These results have been presented to the company that provided us with the case study, and their feedback has been highly positive. They will inspect all the issues or smells identified using the tool to modify those parts.

6.4 Related work

In software maintenance, system overviews can play an important role in identifying reusable components and assessing modularization [Girard and Koschke, 1997]. With a completely different kind of visualization and context we create some visualizations with the same purpose: to investigate the presence of visual patterns and to ease the identification of reusable architectural components.

Also other teams have worked on architecture recovery and validation using, for instance, reflexion models [Murphy *et al.*, 1995a; Murphy and Notkin, 1997]. These models are used to recover architecture by capturing developer knowledge and then manually mapping this knowledge to the source code [Koschke and Simon, 2003; Christl *et al.*, 2005]. Through reflexion models it is possible to expose extra information implicit in the code. Our work exposes information implicit in the different technologies used for building Enterprise applications.

Intensional views check conformance of source code to architectural constraints by means of rules expressed in a dedicated logic programming sublanguage [Mens *et al.*, 2006]. We are unaware of intensional views being used to analyze heterogeneous language projects, or to assess the quality of transactional code.

An interesting study of transactions in EJB has been made at the Vrije Universiteit Brussels [Fabry, 2004]. This work highlights the separation of concerns in the transaction management of JEAs. The problem is caused by the scattered definition of the components involved in a transaction, the operations to perform in case of rollback, and the handling of the rollback exception. The work aimed to improve the separation of concerns by using Aspect-Oriented Programming.

6.5 Conclusion

In this chapter we proposed a solution to the problem of identifying application transactions in JEAs. Transactions aid the application programmer with issues like failure recovery and multi-user programming. Because the definitions of methods and their transaction attributes are decoupled in JEAs, it is difficult for developers and designers to identify whether a method is in a transactional scope or not.

Information unification is one key point. Another one is to present the result of this unification with visualizations of the whole system code to expose anomalies.

Our contribution consists of a technique to expose the transaction scope of the application's classes and to identify related issues. We developed three novel visualizations to expose structural and behavioral anomalies in the definition and use of transaction scope.

The visualizations and the results have been presented to the company that provided us with the case study, and their feedback has been highly positive. The customer checked the anomalies that we identified. They discovered that in the last version of the code all 5 methods starting an unnecessary transaction do so deliberately (for example to obtain a separate transaction scope for logging purposes). The Session Beans accessing the database directly without passing through all the

layers actually perform read operations on the database for visualization purposes unrelated to the business logic, so they decided not to modify them. They also plan a further investigation into the hierarchies outside any transaction scope that actually access the data base to assess eventual problems. They also would like to periodically check their code with these instruments to monitor the status of their application.

The call graphs built from the application analyzed do not use any kind of type approximation. This fact does not affect the techniques and the considerations presented in the paper. Nevertheless, using a different and more accurate call graph on the same application could lead to different results. In particular the *unsafe query* visualization will have more benefits from a more accurate call graph revealing more hierarchies of classes not included in a transaction scope.

The work presented in this chapter focuses on application transactions defined using the deployment descriptor (Container-Managed Transaction Demarcation), but it is possible to define by hand the transaction scope in the code using the Java Transaction API (Bean-Managed Transaction Demarcation). These two definition are implemented in JEAs using EJB version 2.1 [DeMichiel, 2003] and version 3.0 [Linda DeMichiel, 2006] respectively. To extend our analysis on the latest EJB version we would need to identify the transaction scopes using information extracted from the Java Transaction API.

The approach presented in this chapter exploits our first-class description of HAs to analyze application transactions from different points of view. To build the Unsafe Queries visualization we used our approach to map relational database elements to source code elements. This approach is based on the detection of direct accesses on the database defined using the *java.sql* package. This option, however, can be replaced by other strategies to connect database tables with source code entities. In the very specific case of the application transaction problem the database table accessed by a method it is not relevant, however, it could be useful to perform other kind of analyses on the database. For example, we could drive data consistency checking on the database tables that have been accessed outside an application transaction scope. To build the Server Layers visualization we used our approach to analyze architectural layers. By using this approach we could slice the system classes into layers to expose structural violations of the elements involved in a transaction. We could highlight the EJBs located in the wrong layer and detect all the method invocations that jump a layer. Such violations could affect the program comprehension other than be signs of more serious defects in the application code.

Chapter 7

Supporting dependency analysis in HAs

Identifying the connections between software elements is important to understand the behaviour of an application and to perform tasks like impact analysis [Hassan and Holt, 2004] and change propagation [Rajlich, 1997]. Not all the connections between software entities are explicit: some of them are implicitly defined by other dependencies [Zhifeng Yu, 2001; Vanciu and Rajlich, 2010]. Implicit dependencies can be derived from explicit or implicit dependencies. For example, the connection between two classes can be derived from method invocations or attribute accesses. In EAs some connections can be derived from relationships between elements in a different domain [Aryani *et al.*, 2011]. In this scenario the same dependency can be derived from various Heterogeneous sources. For example, dependencies between database tables can be derived from method invocations.

In this chapter we present our approach to support the analysis of implicit dependencies between HA elements. Such an approach is based on a model which reifies the concept of vertical and horizontal relationships applied to graph nodes. Vertical relationships are used to move from a lower to a higher level of abstraction, while horizontal relationships are use to move between elements at a similar level of abstraction. The semantic of these relationships is not strict and can be adapted to follow the user needs. As a consequence, this model can also describe any graph-based data structure. Implicit dependencies are inferred on top of this generic model. We identified three relevant properties that implicit dependencies can have from the perspective of software analysis: (1) the same dependency can be computed by multiple explicit dependencies, (2) implicit dependencies can be at different abstraction levels than the explicit dependencies used to infer them, (3) an implicit dependency should expose the explicit dependencies used to infer it. In this chapter we also formalize how the implicit dependencies are inferred starting from a set of given associations.

The idea of aggregating or mapping lower level relationships to higher level relationships was formalized by Feijs *et al.* [Feijs *et al.*, 1998] and takes the name of "lift-

ing" theory. Feijs *et al.* formalized this theory to support the analysis of software architectures. The authors applied relational algebra on graphs to validate architectural constrains on existing software systems. A previous approach that inspired the lifting theory was proposed by Murphy *et al.* [Murphy *et al.*, 1995b]. The authors attempted to map the entities of a software reflexion model on top of real source code elements. Considering that more than one source code element can be mapped on the same reflexion model entity, the approach from Murphy *et al.* can be intended as a lifting technique. Our approach is similar to the lifting technique proposed by Feijs *et al.*, however, our derivation process is based on our own model rather than on a graph. In addition, our technique is meant to support the analysis of software dependencies so it is not limited to the validation of architectural constraints.

To validate our approach we developed the Carrack inference engine and we applied it to the case study presented in Section 4.3.2 to verify constraints between architectural associations.

Carrack is implemented in Pharo[1] Smalltalk. Readers unfamiliar with the syntax of Smalltalk might want to read the code examples in the remainder of this dissertation aloud and interpret them as normal sentences. An invocation to a method named `method:with:`, using two arguments looks like: `receiver method: arg1 with: arg2`. The semicolon separates messages that are sent to the same receiver. For example, `receiver method1: arg1; method2: arg2` sends the messages `method1:` and `method2:` to `receiver`. Other syntactic elements of Smalltalk are: the dot to separate statements: `statement1. statement2`; square brackets to denote code blocks or anonymous functions: `[ statements ]`; single quotes to delimit strings: `'a string'`; and double quotes to delimit comments: `"comment"`. The caret `^` returns the result of the following expression.

7.1 The Carrack Meta-Model

Figure 7.1 shows the meta-model we use to support dependency analysis in HAs as it is implemented in Carrack. A `CarrackEntity` has a one to one relation with the model elements we want to infer relations for. A `CarrackEntity` can contain other carrack entities and can be associated to one or more carrack entities. A `CarrackContainment` is used to define what we call *vertical relations* among carrack nodes. A `CarrackAssociation` is used to define what we call *horizontal relations* among carrack nodes. The semantics of containments and associations depend on how the user defines them. For example, we can define as an association relationships like class inheritances, and as a containment relationships like method invocations, or vice

1 `http://www.pharo-project.org/home`

versa. Each Carrack model element contains a reference to the original model element it represents. Therefore, during the analysis of the Carrack model we can for example, select the Carrack model nodes that contain elements of a specific type.

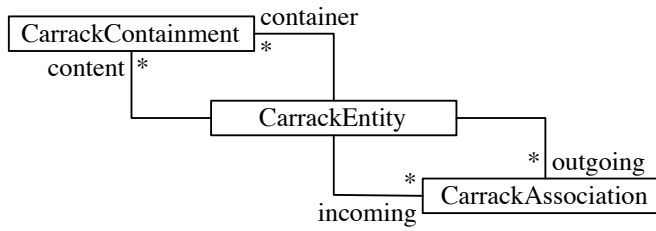
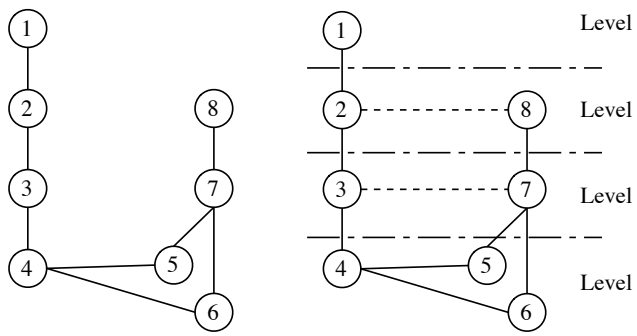


Figure 7.1: Carrack meta-model

Carrack can support the analysis of implicit connections between model elements which can differ in semantic and abstraction level by exploiting its internal meta-model. For example, we could infer the implicit dependencies between database tables starting from explicit dependencies between software elements.

7.2 Derived Dependency Inference

In this section we will define how we infer implicit associations within a carrack model, and we will demonstrate this definition on the example graph shown in Figure 7.2a.



(a) Sample graph **(b)** Graph described in Carrack

Figure 7.2: Sample graph

The code snippet in Listing 7.1 shows the script that we would execute to create a carrack model from the graph in Figure 7.2a. In line 4 we inform the model builder about the elements composing the carrack model by using the method *nodes:*. We then define which of the graph edges we consider to be a containment relationships using the method *defineContainmentFrom:to:on:* in line 6–9. The containment relationships will conceptually organize the graph into levels as shown in Figure 7.2b. The third step is to define which of the graph edges we consider to be associations using the method *defineAssociationFrom:to:on:* in line 10–13. Finally, in Line 15 we request the model builder to instantiate the carrack model based on our definition.

```
1  | builder |
2  builder := CAKModelBuilder new.
3
4  builder nodes: #(1 2 3 4 5 6 7 8).
5
6  builder
7     defineContainmentFrom: #first
8     to: #second
9     on: {#(3 4).#(2 3).#(1 2).#(7 6).#(7 5).#(8 7)}.
10 builder
11    defineAssociationFrom: #first
12    to: #second
13    on: {#(4 5).#(4 6)}.
14
15 ^builder create model
```

Listing 7.1: Possible carrack model definition on a graph with eight elements implemented in Smalltalk

The carrack model creation involves 2 main steps: during the first step all the elements and the relations defined by the user with the model builder API will be created. In the second step the builder will infer all the implicit associations among the elements that have not been defined by the user.

The following definition describes how we infer associations within a carrack model:

$$(\alpha(A))^* \tag{7.1}$$

where:

$$N = \text{all Carrack nodes} \tag{7.2}$$
$$C \subseteq N \times N \text{ are containments} \tag{7.3}$$
$$A \subseteq N \times N \text{ are associations} \tag{7.4}$$
$$\alpha(a,b) = \{(c,d) | (a,b) \in A \wedge (c,a) \in C \wedge (d,b) \in C\} \tag{7.5}$$

N is the set containing all the nodes of a carrack model. C represents the containment relations between nodes. A represents the association relation between two nodes. Finally, $\alpha(a,b)$ returns a node pair (c,d) so that c contains a, d contains b and $(a,b) \in A$.

We can better exemplify how the carrack model creation process works by making use of our sample graph in Figure 7.2a. When we request the model builder to instantiate the carrack model (line 15 in Listing 7.1): (1) it will instantiate one `CarrackEntity` for each of the graph nodes specified in line 4. (2) it will create a `CarrackContainment` relation among the elements as specified in lines 6–9. (3) it will create a `CarrackAssociation` among the entities as specified in lines 10–13. Each of the entities created in this steps will have a link to the starting model elements used in the model definition script. (4) Finally, the builder will iteratively infer all the implicit dependencies between the model elements starting from the user defined associations.

7.3 Analysis of Architectural Dependencies with Carrack

In Section 4.3.2 we described how we used our meta-model for software architectures to validate constraints defined on architectural associations. Our goal was to investigate the presence of four different types of undesired architectural associations:

1. Associations from the business layer to the presentation layer.
2. Associations from global components to local components.
3. Associations between global components.
4. Associations between the business layers of different functions.

Our industrial partner requested us to derive dependencies between architectural components based on method invocations and attribute accesses.

In our experience, the derivation process has typically to be implemented in an *ad hoc* manner. This approach has two main drawbacks: (1) the same dependency description must be defined multiple times, and (2) reusing dependency descriptions is not always possible due to the functional detail of each case.

In this section we describe how we can overcome these two drawbacks by using our approach to support the analysis of derived dependencies.

Listing 7.2 shows the script we used to infer architectural level dependencies based on attribute accesses in the industrial system we analyzed.

```
| builder mooseModel |

mooseModel := MooseModel root first.
builder := CAKModelBuilder new.

builder nodes: mooseModel allCSComponents.
builder nodes: mooseModel allModelClasses.
builder nodes: mooseModel allModelMethods.
builder nodes: mooseModel allAttributes.

builder
   defineContainmentsFrom: #containerComponent
   to: #containedElement
   on: mooseModel allComponentContainment.
builder defineContainmentFrom: #yourself to: #methods on: mooseModel allModelClasses.
builder defineContainmentFrom: #yourself to: #attributes on: mooseModel allModelClasses.
builder defineAssociationFrom: #accessor to: #variable on: mooseModel allAccesses.

builder create.
^builder model.
```

Listing 7.2: carrack script to infer architectural level dependencies base on attribute accesses

Table 7.1 shows several metrics on the output of the script in Listing 7.2. The computation of the script took $\sim 13s$ [2] to generate a carrack model of $167,086$ elements. These elements include: $60,542$ carrack entities, $52,502$ containments and $54,042$ associations, $3,139$ of which were inferred by lower level dependencies. The dependencies inferred between architectural elements amounted to 136. The dependencies that were not inferred between architectural elements were inferred between system classes. The time required to infer the dependencies was $\sim 4.2s$, or approximately 30% of the total computation time.

2 The computation was done on a MacBookPro with 8GB of RAM and a 2.66Ghz Intel CPU with 4 cores

Model Size	$167,086$
Execution Time	$\sim 13s$
Nodes	$60,542$
Containments	$52,502$
Associations	$54,042$
Inferred Associations	$3,139$
Inferred Associations between architectural elements	136
Inference elapsed time	$\sim 4.2s$

Table 7.1: Metrics on the output of Listing 7.2

Listing 7.3 shows the script we used to infer the associations between the same architectural components used in Listing 7.2, based this time on method invocations.

```
1  | builder mooseModel |
2
3  mooseModel := MooseModel root first.
4  builder := CAKModelBuilder new.
5
6  builder nodes: mooseModel allCSComponents.
7  builder nodes: mooseModel allModelClasses.
8  builder nodes: mooseModel allModelMethods.
9
10 builder
11    defineContainmentsFrom: #containerComponent
12    to: #containedElement
13    on: mooseModel allComponentContainment.
14 builder defineContainmentsFrom: #yourself to: #methods on: mooseModel allModelClasses.
15 builder defineAssociationsFrom: #sender to: #candidates on: mooseModel allInvocations.
16
17 builder create.
18 ^builder model
```

Listing 7.3: Carrack script to infer architectural level dependencies based on method invocations

The only differences between Listing 7.3 and Listing 7.2 are: (1) the missing definition of source code attributes and their connections, as they are not relevant in this case (lines 9 and 16 in Listing 7.2), and (2) the different definition of node associations (line 15 in Listing 7.3). The need for such minor modifications between the two scripts is an example of the flexibility of our approach.

Table 7.2 lists several metrics on the output of the script in Listing 7.3. The computation of the script took $\sim 9s$ to generate a carrack model of $144,251$ elements. These

elements include: $38,755$ carrack entities, $37,474$ containments and $68,022$ associations, $8,938$ of which were inferred by lower level dependencies. The amount of dependencies inferred between architectural elements was 292. The time spent to infer the dependencies was $\sim 3.5s$, or approximately 40% of the total computation time. By comparing Table 7.1 and Table 7.2 it is clear that the total amount of elements of the model is lower in the second case. This is due to the fact that the definitions of class attributes and accesses have been removed in Listing 7.3, since they were not needed anymore. The same argument can be used to explain why the computation of the script in Listing 7.3 was faster than that of the script in Listing 7.2. The number of inferred associations sensibly increases due the presence in the code of a number of method invocations higher than the number of accesses.

Model Size	$144,251$
Execution Time	$\sim 9s$
Nodes	$38,755$
Containments	$37,474$
Associations	$68,022$
Inferred Associations	$8,938$
Inferred Associations between architectural elements	292
Inference elapsed time	$\sim 3.5s$

Table 7.2: Metrics on the output of Listing 7.3

The output of the scripts in Listing 7.2 and Listing 7.3 are two distinct carrack models that we used to validate the four architectural constraints listed at the beginning of this section. Listing 7.4 shows the query used to check global component isolation (presented in Listing 4.1), adapted to use a carrack model.

```
| componentAssociations res |

componentAssociations := (self model allEntitiesKindOfType: CSComponent)
     allOutgoingAssociations.

res := componentAssociations
   select: [ :association |
      association source model isGlobal
         and: [
            association target model isGlobal
               and: [ association source model ~= association target model ] ] ].
res := self applyFiltersOn: res.
^ res
```

Listing 7.4: Query to detect connections among global components using carrack

The query in Listing 7.4 is almost identical to the query in Listing 4.1. It has, however, the significant advantage that it can be reused on any carrack model that contains the architectural components the query is designed for. In fact, we used this query to validate the same constraint on both the carrack models generated by the scripts in Listing 7.2 and Listing 7.3 respectively. The results obtained by using Carrack on the banking system we analyzed, were exactly the same as those we obtained by hard-coding the inference logic and the queries in our model for HAs (Section 4.3.2).

7.4 Conclusions

Software dependency analysis has always been important to evaluate the propagation of changes and, in general, to understand the interaction between application components. Dependencies between software system elements are not necessarily explicit as some of them are implicitly defined by other dependencies. We argue that implicit dependencies have at least three important properties from the perspective of software analysis: (1) the same dependency can be computed by multiple explicit dependencies, (2) implicit dependencies can be at a different abstraction level than the explicit dependencies used to infer them, (3) an implicit dependency should expose the explicit dependencies used to infer it.

To support the analysis of derived dependencies in HAs, we provided an approach based on a meta-model that reifies the concepts of horizontal and vertical relationships which fulfill these three properties. Our approach relies on an internal first-class description of the system to analyze, thus it makes possible to infer associations between semantically different model elements. We also defined a process to infer derive dependencies within graph-based structures.

To validate our approach we developed the Carrack inference engine. We used Carrack to derive and analyze the dependencies between the architectural components of an industrial banking system. The architectural reconstruction we performed on the banking system followed a bottom-up approach. In this scenario the architectural connections were derived starting from source code level dependencies.

By using Carrack we were able to script, and thus to ease, the definition of the architectural dependencies in our case study. We were also able to decouple and generalize the definition of architectural constraints from the original architectural model.

Our approach to support the analysis of derived dependencies needs to be validated on larger case studies, in spite of the positive results we described in this section. We

still need to identify, and possibly formalize, the exact boundaries of applicability of our approach. Finally, we need to improve Carrack to make it even more performant and to exploit completely the potentiality of our approach by enabling, for example, the analysis of ternary relationships.

Chapter 8

Conclusions

Most modern software systems are built using various frameworks, technologies and languages to address different application requirements. We call such systems heterogeneous applications because the information regarding their structure and behaviour can be spread across various components. This composition of elements increases the complexity of the analysis of heterogeneous applications. In this context, applying existing reverse engineering and quality assurance techniques developed for homogeneous applications is not enough. We argue that to overcome this problem and to support various kinds of analyses on HAs we need a unified representation of the heterogeneous elements composing such systems. We also need to support the analysis of explicit and implicit dependencies between elements at different levels of abstraction.

The key contributions described in this dissertation are the following:

- We presented (Chapter 3) our first-class description of heterogeneous applications that takes into account: architecture, relational databases and specific JEA technologies like EJBs. We validated these three descriptions on three case studies: We performed (Chapter 4) architectural understanding and constraint validation on an industrial banking system by exploiting our first-class description of software architectures. We defined (Chapter 5) a unified approach to support the co-evolution of applications and their persistent data structures based on our meta-model for relational databases. We analyzed (Chapter 6) the scope of application transactions in JEAs exploiting our fist-class description of EJBs, and we demonstrated how we could handle other facets of this problem using our meta-models for HAs as a whole.
- We introduced (Chapter 7) our approach to analyze derived dependencies in HAs. The approach exploits a model that can describe any graph-based data structure, thus can be used in combination with our model for HAs. The derived dependencies are inferred on top of this generic model. We validated this approach by inferring the dependencies between architectural elements

of an industrial banking system starting from the associations between its source code elements.

Our meta-model for HAs is meant to be generic enough to be applied on different kinds of heterogeneous applications. The only part specific for JEAs is the description of EJBs which can be excluded from the rest of the meta-model if our approach is applied on a non Java enterprise system. We implemented our approach in a tool called MooseEE as an extension of the Moose platform for software and data analysis. Moose has been chosen for two main reasons: (1) to use the FAMIX meta-model for object-oriented and procedural languages and, (2) to take advantage of the engines that compose the Moose eco-system which are useful to script software visualizations, reports, charts *etc.*

Our approach to analyze derived dependencies between the elements composing HAs has been implemented in a tool called Carrack. The reification of implicit dependencies can support the user in tasks like impact change analysis and architectural validation and understanding, not only in the context of HAs.

8.1 Future Work

In this section we provide an overview of further research and practical next steps related to the implementation of our approach.

8.1.1 Research Directions

- Architectural models that describe architectures as a set of components and connectors between these components are too simplistic. Nowadays applications are composed of various elements that can be structured in layers and they can also be distributed. The component-connector model is meant to describe such elements, however, needs to be specialized to represent a specific architecture. We need to identify more specific elements to describe architectures, thus providing analysis tools and techniques with elements characterized by unambiguous semantics. Furthermore, the focus of architectural descriptions based on components and connectors is on structural and technological elements, however, developers and engineers are not the only stakeholder of an application. Aspects related to the system domain or the business context of an application must be identified and reified within architectural models. These models can then be used by analysis techniques to fully support architectural understanding from the points of view of the different stakeholders involved in an application life cycle. The domain and

business aspects we are referring to comprise, for example, functional and non-functional requirements such as the development costs of a component or the response time of a function.

- The process we use to create our homogeneous description of heterogeneous systems consists in importing information from multiple sources and merging this information within a unified model. Each information source is parsed and the required information is harvested by a fact extractor applied to the results of the parsing phase. A number of parsers for several languages and data formats are already present, however, these parsers usually create an exact representation of the parsed elements, while a more abstract representation would be more suitable to collect information for software analysis purposes. Even though a lot of research effort has been spent in this area we believe that more effort should be spent in the direction of automatic creation of parsers and of intermediate representations.

- A standard and largely applied technique to make analyses reusable is to based them on models that describe the software systems to analyze rather than on the systems themselves. In our experience, however, analyses that address generic problems, and that cannot be adapted to work on specific implementations, are not used on real systems or the results of these analyses are not taken into account by analysts. This happens because software analysis techniques and tools are often required to address very specific problems that can hardly be solved using general purpose analyses. We argue that to support software analyses in general, and HAs in particular, we need to invest more research effort to provide reverse engineers with techniques and tools to quickly craft custom analyses and tools.

8.1.2 Practical Steps

Our first-class representation of heterogeneous applications is not complete and needs to be extended:

- The architecture of heterogeneous applications is not simply composed of components and connectors. More detailed elements are required to effectively describe the real architecture of heterogeneous systems. Our unified representation needs to be enriched with these more specific elements. For example, we need to reify structural elements like the various tiers composing an application, but also nonstructural elements like “product” or “feature”.

- We proposed a meta-model to represent relational databases since they are still largely used in industry, however, they are not the only way to make

data persist. A first-class representation of the other persistent data structures is needed to achieve our goal to have a uniform representation of heterogeneous systems. For example, the structure of object-oriented or XML databases needs to be model to enable analyses on these types of data storage systems. Another example is represented by data warehouses. Analyses on these databases might benefit from the reification of logical elements like "dimension" or "fact" that are used to arrange the data into hierarchical groups.

- The FAMIX model is able to represent software languages following the object-oriented or procedural paradigms. A model that can reify also the other paradigms used by software systems will provide a good base to represent the multi-language nature of heterogeneous applications.

Our approach to infer derived dependencies between the elements composing heterogeneous applications is good enough to serve its purpose, however, our implementation does not fully exploit its potential. Carrack infers dependencies starting from the direct connections defined by user. Direct connections, however, are not the only ones that can be interesting for software analyses [Zhifeng Yu, 2001; Vanciu and Rajlich, 2010; Aryani *et al.*, 2011]. For example, Ternary or N-ary dependencies are also important to achieve impact change propagation or other dependency analyses not only in heterogeneous applications.

Appendix A

Getting Started

This appendix gives instructions on how to install MooseEE and Carrack in a Moose image.

Downloading MooseEE

1. Get a Moose image from `http://www.moosetechnology.org/download`.
2. Execute the following lines to load MooseEE in the Moose image:

```
Gofer new
   squeaksource: 'MooseJEE';
   package: 'ConfigurationOfMooseJEE';
   load.
(Smalltalk at: #ConfigurationOfMooseJEE) perform: #loadDefault
```

MooseEE have been initially developed under the name of MooseJEE this is why the repository still carry that name.

Downloading Carrack

1. Get a Pharo image from `http://www.pharo-project.org/` or use a Moose image with MooseEE loaded as explained in the previous section.
2. Execute the following lines to load Carrack in the image:

```
Gofer new
   squeaksource: 'Carrack';
   package: 'ConfigurationOfCarrack';
   load.
(Smalltalk at: #ConfigurationOfCarrack) perform: #loadDefault.
```

Appendix B

Moose platform

Our description of HAs presented in Chapter 3 has been implemented in MooseEE as an extension of the Moose platform for software and data analyses. The general Moose workflow shown in Figure B.1 reflects the well-known extract-abstract-present cycle described by Tilley *et al.* [Tilley *et al.*, 1996].

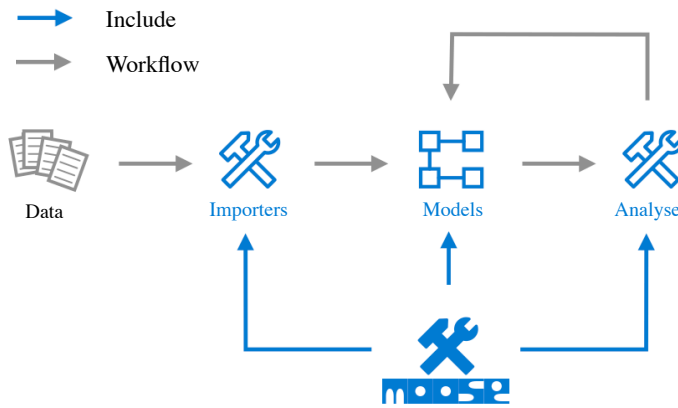


Figure B.1: General workflow working with Moose

Starting from some data, intended as any structures containing objects, properties and relations, we extract the information we need using importers. Moose provides the user with importers for data from various sources and in various formats. The importers can be internal (*e.g.*, the importers for Smalltalk code, XML and MSE files), or external (*e.g.*, the importers for Java, C and C++ are implemented in *inFamix*[1]) to the Moose platform.

The imported data is stored in models. At the core of Moose is the language-independent FAMIX meta-model [Tichelaar *et al.*, 2000] which can describe the static

1 http://www.intooitus.com/products/infamix/

structure of software systems. Figure B.2 shows the core of the FAMIX meta-model as is implemented in Smalltalk. FAMIX contains all the elements composing object-oriented and procedural languages (*i.e.*, classes, methods, attributes, namespaces *etc.*) as well as all the associations among them (*i.e.*, inheritances, invocations, accesses *etc.*). The most generic class in the hierarchy is `Entity`, while its direct subclass `SourceEntity` is the most generic representation of source code elements. On top of the abstracted data we can start performing various kinds of analysis by means of metrics, queries, visualizations *etc.*

We choose Moose as the platform for our implementation for the very reason that it is a platform and not a tool. Moose helps the user to build custom tools. In particular, Moose helps the user to:

- build new importers for new data sets,
- define new models to store the data, and
- create new analysis algorithms and tools such as: graph visualizations, charts, queries, browsers and reporting tools.

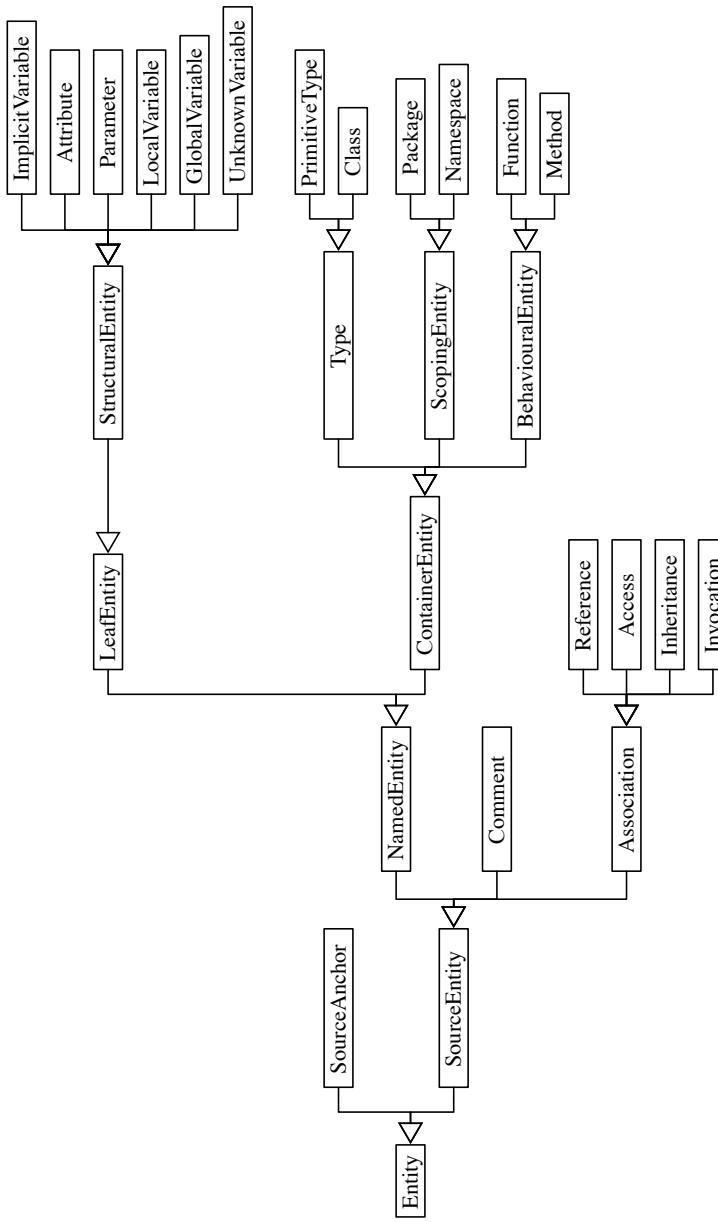


Figure B.2: The FAMIX 3.0 meta-model Core to represent software languages

Appendix C

Bibliography

[Alur *et al.*, 2001] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Pearson Education, 2001.

[Arcelli Fontana and Zanoni, 2011] Francesca Arcelli Fontana and Marco Zanoni. A tool for design pattern detection and software architecture reconstruction. *Information Sciences*, 181(7):1306–1324, April 2011.

[Arcelli Fontana *et al.*, 2010] Francesca Arcelli Fontana, Gianluigi Viscusi, and Marco Zanoni. Unifying software and data reverse engineering - a pattern based approach. In Maria Virvou José A. Moinhos Cordeiro and Boris Shishkov, editors, *ICSOFT 2010 - Proceedings of the Fifth International Conference on Software and Data Technologies*, volume 2, pages 208–213, Athens, Greece, July 2010. SciTePress.

[Arisholm *et al.*, 2004] E. Arisholm, L.C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *Software Engineering, IEEE Transactions on*, 30(8):491–506, August 2004.

[Armstrong *et al.*, 2005] Eric Armstrong, Jennifer Ball, Stephanie Bodoff, Debbie Bode Carson, Ian Evans, Dale Green, Kim Haase, and Eric Jendrock. The J2EE 1.4 tutorial, December 2005.

[Aryani *et al.*, 2011] Amir Aryani, Fabrizio Perin, Mircea Lungu, Abdun Naser Mahmood, and Oscar Nierstrasz. Can we predict dependencies using domain information? In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE 2011)*, October 2011.

[Batini *et al.*, 2006] Carlo Batini, Daniele Barone, Manuel F. Garasi, and Gianluigi Viscusi. Design and use of er repositories: Methodologies and experiences in egovernment initiatives. In *Proceedings of 25th International Conference on Conceptual Modeling, ER*, volume 4215 of *Lecture Notes in Computer Science*, pages 399–412. Springer, November 2006.

[Batini *et al.*, 2011] Carlo Batini, Marco Comerio, Enrica Pasqua, and Gianluigi Viscusi. Repositories of conceptual schemas: Concepts, constructs, methods and quality dimensions. In Giansalvatore Mecca and Sergio Greco, editors, *Proceedings of the Nineteenth Italian Symposium on Advanced Database Systems*, pages 381–388, June 2011.

[Binkley and Harman, 2004] David Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105 – 178, 2004.

[Binkley, 2007] David Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.

[Bohner and Arnold, 1996] S. A Bohner and R.S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[Canfora *et al.*, 2011] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and challenges in software reverse engineering. *Commun. ACM*, 54(4):142–151, April 2011.

[Carey *et al.*, 2012] Michael J. Carey, Nicola Onose, and Michalis Petropoulos. Data services. *Commun. ACM*, 55(6):86–97, June 2012.

[Chidamber and Kemerer, 1991] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '91, pages 197–211, New York, NY, USA, 1991. ACM.

[Chidamber and Kemerer, 1994] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[Chikofsky, 1996] Elliot J. Chikofsky. The necessity of data reverse engineering. foreword for peter aiken's data reverse engineering, 1996.

[Christl *et al.*, 2005] Andreas Christl, Rainer Koschke, and Margaret-Anne Storey. Equipping the reflexion method with automated clustering. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 89–98, 2005.

[Cleary and Exton, 2007] B Cleary and Chris Exton. Assisting concept location in software comprehension. In *Proceedings of 19th Annual Psychology of Programming Workshop (PPIG 07)*, Joensuu, Finland, July 2007.

[Cleve *et al.*, 2006] Anthony Cleve, Jean Henrard, and Jean-Luc Hainaut. Data reverse engineering using system dependency graphs. In *WCRE 06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 157 –166, October 2006.

[Cleve *et al.*, 2010a] Anthony Cleve, Anne-France Brogneaux, and Jean-Luc Hainaut. A conceptual approach to database applications evolution. In *Proceedings of the 29th international conference on Conceptual modeling*, ER'10, pages 132–145, Berlin, Heidelberg, 2010. Springer-Verlag.

[Cleve *et al.*, 2010b] Anthony Cleve, Tom Mens, and Jean-Luc Hainaut. Data-intensive system evolution. *Computer*, 43:110–112, 2010.

[Cornelissen *et al.*, 2009] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[D'Ambros *et al.*, 2009] Marco D'Ambros, Michele Lanza, and Romain Robbes. On the relationship between change coupling and software defects. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 135–144, Washington, DC, USA, 2009. IEEE Computer Society.

[Davis and Aiken, 2000] Kathi Hogshead Davis and Peter H. Aiken. Data reverse engineering: A historical survey. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, WCRE '00, pages 70–, Washington, DC, USA, 2000. IEEE Computer Society.

[Deissenboeck *et al.*, 2005] F. Deissenboeck, M. Pizka, and T. Seifert. Tool support for continuous quality assessment. In *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*, pages 127 –136, September 2005.

[DeMichiel, 2003] Linda G. DeMichiel. Enterprise JavaBeans specification, version 2.1, November 2003.

[Ducasse and Pollet, 2009] Stéphane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE TSE: Transactions on Software Engineering*, 35(4):573–591, July 2009.

[Ebert *et al.*, 1996] Jürgen Ebert, Andreas Winter, Peter Dahm, Angelika Franzke, and Roger Süttenbach. Graph based modeling and implementation with eer / gral. In *Proceedings of the 15th International Conference on Conceptual Modeling*, ER '96, pages 163–178, London, UK, UK, 1996. Springer-Verlag.

[Ebert *et al.*, 2002] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO — generic understanding of programs, an overview. Fachberichte Informatik 7–2002, Universität Koblenz-Landau, 2002.

[Eichberg *et al.*, 2008] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies.

In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 391–400, New York, NY, USA, 2008. ACM.

[Fabry, 2004] Johan Fabry. Transaction management in EJBs: Better separation of concerns with AOP. In Y. Coady and D. Lorenz, editors, *Proc. of the 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 20–25, March 2004.

[Feijs *et al.*, 1998] L. Feijs, R. Krikhaar, and R. Van Ommering. A relational approach to support software architecture analysis. *Softw. Pract. Exper.*, 28(4):371–400, April 1998.

[Fowler, 2005a] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 75 Arlington Street Suite 300, Boston MA, 02116 USA, 2005.

[Fowler, 2005b] Martin Fowler. *Patterns of Enterprise Application Architecture*, chapter 8.4: Other layering schemes. Addison Wesley, 2005.

[Gall *et al.*, 2003] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, IWPSE '03, pages 13–, Washington, DC, USA, 2003. IEEE Computer Society.

[Gall *et al.*, 2008] C.S. Gall, S. Lukins, L. Etzkorn, S. Gholston, P. Farrington, D. Utley, J. Fortune, and S. Virani. Semantic software metrics computed from natural language design specifications. *Software, IET*, 2(1):17–26, February 2008.

[Garcia-Molina *et al.*, 2011] Hector Garcia-Molina, Georgia Koutrika, and Aditya Parameswaran. Information seeking: convergence of search, recommendations, and advertising. *Commun. ACM*, 54(11):121–130, November 2011.

[Garlan, 2000] David Garlan. Software architecture: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 91–101, New York, NY, USA, 2000. ACM.

[Gethers and Poshyvanyk, 2010] Malcom Gethers and Denys Poshyvanyk. Using relational topic models to capture coupling among classes in object-oriented software systems. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, September 2010. IEEE Computer Society.

[Girard and Koschke, 1997] Jean-Francois Girard and Rainer Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *ICSM*. IEEE Press, 1997.

[Group, 2004] Object Management Group. Meta object facility (MOF) 2.0 core final adopted specification. Technical report, Object Management Group, 2004.

[Hainaut *et al.*, 2000] Jean L. Hainaut, Jean Henrard, Jean M. Hick, Didier Roland, and Vincent Englebert. The nature of data reverse engineering. In *Proc. of Data Reverse Engineering Workshop 2000 (DRE'2000)*. Zurich Univ. Publish., 2000.

[Hammad *et al.*, 2009] M. Hammad, M.L. Collard, and J.I. Maletic. Automatically identifying changes that impact code-to-design traceability. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 20 –29, May 2009.

[Hammond *et al.*, 2008] Jeffrey S. Hammond, Noel Yuhanna, M. Gilpin, and D. D'silva. Market overview: Enterprise data modeling: A steady state market prepares to enter a transformational new phase, October 2008.

[Harman *et al.*, 2009] Mark Harman, David Binkley, Keith Gallagher, Nicolas Gold, and Jens Krinke. Dependence clusters in source code. *ACM Trans. Program. Lang. Syst.*, 32:1:1–1:33, November 2009.

[Hassan and Holt, 2004] Ahmed Hassan and Richard Holt. Predicting change propagation in software systems. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 284–293, Los Alamitos CA, September 2004. IEEE Computer Society Press.

[Hassan and Holt, 2006] Ahmed E. Hassan and Richard C. Holt. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Softw. Engg.*, 11(3):335–367, September 2006.

[Hassoun *et al.*, 2004] Youssef Hassoun, Roger Johnson, and Steve Counsell. A dynamic runtime coupling metric for meta-level architectures. *Software Maintenance and Reengineering, European Conference on*, 0:339, 2004.

[Henrard *et al.*, 2002] J. Henrard, J.-M. Hick, P. Thiran, and J.-L. Hainaut. Strategies for data reengineering. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 211 – 220, 2002.

[Henrard *et al.*, 2007] J. Henrard, D. Roland, A. Cleve, and J.-L. Hainaut. An industrial experience report on legacy data-intensive system migration. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 473 –476, October 2007.

[Hindle and Jordan, 2004] Daniel Germáin Abram Hindle and Norman Jordan. Visualizing the evolution of software using softchange. In *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE 2004)*, pages 336–341, New York NY, 2004. ACM Press.

[Horwitz *et al.*, 1990] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, January 1990.

[IEEE, 2000] IEEE. Ieee recommended practice for architectural description for software-intensive systems. Technical report, The Architecture Working Group of the Software Engineering Committee, October 2000.

[Kagdi *et al.*, 2007] H. Kagdi, J.I. Maletic, and B. Sharif. Mining software repositories for traceability links. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, pages 145 –154, June 2007.

[Keller, 1997] Wolfgang Keller. Mapping objects to tables - a pattern language. In *Proc. Of European Conference on Pattern Languages of Programming Conference EuroPLOP '97*, 1997.

[Keller, 1998] Wolfgang Keller. Object / relational access layers - a roadmap, missing links and more patterns. In *In Proceeding of EuroPlop 1998, Irsee*, pages 1–25, 1998.

[Knodel and Popescu, 2007] J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. In *Software Architecture, 2007. WICSA '07. The Working IEEE/IFIP Conference on*, page 12, January 2007.

[Knodel *et al.*, 2006] Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. Static evaluation of software architectures. In *CSMR'06*, pages 279–294, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[Koschke and Simon, 2003] Rainer Koschke and Daniel Simon. Hierarchical reflexion models. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, page 36. IEEE Computer Society, 2003.

[Kullbach *et al.*, 1998] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program comprehension in multi-language systems. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, WCRE '98, pages 135–, Washington, DC, USA, 1998. IEEE Computer Society.

[Lanza and Marinescu, 2006] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[Linda DeMichiel, 2006] Michael Keith Linda DeMichiel. JSR 220: Enterprise JavaBeans specification, version 3.0, May 2006.

[Linos *et al.*, 2003] P.K. Linos, Zhi hong Chen, S. Berrier, and B. O'Rourke. A tool for understanding multi-language program dependencies. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 64 – 72, May 2003.

[Linos *et al.*, 2007] Panos Linos, Whitney Lucas, Sig Myers, and Ezekiel Maier. A metrics tool for multi-language software. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, SEA '07, pages 324–329, Anaheim, CA, USA, 2007. ACTA Press.

[Luqi, 1990] Luqi. A graph model for software evolution. *IEEE Trans. Softw. Eng.*, 16(8):917–927, August 1990.

[Marinescu and Jurca, 2006] Cristina Marinescu and Ioan Jurca. A meta-model for enterprise applications. In *SYNASC '06: Proceedings of the Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 187–194, Washington, DC, USA, 2006. IEEE Computer Society.

[Marinescu, 2006] Cristina Marinescu. Identification of design roles for the assessment of design quality in enterprise applications. In *Proceedings of International Conference on Program Comprehension (ICPC 2006)*, pages 169–180, Los Alamitos CA, 2006. IEEE Computer Society Press.

[Marinescu, 2007a] Cristina Marinescu. Discovering the objectual meaning of foreign key constraints in enterprise applications. *Reverse Engineering, Working Conference on*, 0:100–109, 2007.

[Marinescu, 2007b] Cristina Marinescu. Identification of Relational Discrepancies between Database Schemas and Source-Code in Enterprise Applications. In *Symbolic and Numeric Algorithms for Scientific Computing, 2007. SYNASC. International Symposium on*, pages 93–100, September 2007.

[Medvidovic and Taylor, 2000] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[Medvidovic *et al.*, 2007] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. Moving architectural description from under the technology lamppost. *Information and Software Technology*, 49(1):12–31, 2007.

[Mens and Demeyer, 2008] Tom Mens and Serge Demeyer, editors. *Software Evolution*. Springer, 2008.

[Mens and Kellens, 2006] Kim Mens and Andy Kellens. IntensiVE, a toolsuite for documenting and checking structural source-code regularities. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10 pp. –248, mar 2006.

[Mens *et al.*, 2006] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Coevolving code and design with intensional views — a case study. *Journal of Computer Languages, Systems and Structures*, 32(2):140–156, 2006.

[Meyer *et al.*, 2006] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[Mian and Hussain, 2008] Natash Ali Mian and Tauqeer Hussain. Database reverse engineering tools. In *Proceedings of the 7th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, SEPADS'08, pages 206–211, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS).

[Mirarab *et al.*, 2007] S. Mirarab, A. Hassouna, and L. Tahvildari. Using bayesian belief networks to predict change propagation in software systems. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, pages 177 –188, June 2007.

[Murphy and Notkin, 1997] Gail C. Murphy and David Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 8:29–36, 1997.

[Murphy *et al.*, 1995a] Gail Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[Murphy *et al.*, 1995b] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '95, pages 18–28, New York, NY, USA, 1995. ACM.

[Murphy *et al.*, 2001] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27(4):364–380, April 2001.

[Nierstrasz *et al.*, 2005] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York, NY, USA, September 2005. ACM Press. Invited paper.

[Perin *et al.*, 2010] Fabrizio Perin, Tudor Gîrba, and Oscar Nierstrasz. Recovery and analysis of transaction scope from scattered information in Java enterprise applications. In *Proceedings of International Conference on Software Maintenance 2010*, September 2010.

[Poshyvanyk and Marcus, 2006] Denys Poshyvanyk and Andrian Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of the 22nd*

IEEE International Conference on Software Maintenance, pages 469–478, Washington, DC, USA, 2006. IEEE Computer Society.

[Poshyvanyk *et al.*, 2006] Denys Poshyvanyk, Andrian Marcus, Vaclav Rajlich, Yann-Gael Gueheneuc, and Giuliano Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 137–148, Washington, DC, USA, 2006. IEEE Computer Society.

[Poshyvanyk *et al.*, 2009] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, February 2009.

[Rajlich, 1997] Vaclav Rajlich. A model for change propagation based on graph rewriting. In *Proceedings of the International Conference on Software Maintenance*, ICSM '97, pages 84–91, Washington, DC, USA, 1997. IEEE Computer Society.

[Raza *et al.*, 2006] Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus - a tool suite for program analysis and reverse engineering. In *Reliable Software Technologies - Ada-Europe 2006*, pages 71–82. LNCS (4006), June 2006.

[Renggli *et al.*, 2010] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010.

[Shaw and Garlan, 1996] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[Silva, 2011] Josep Silva. A vocabulary of program-slicing based techniques. *ACM Computing Surveys*, 2011.

[Spinellis, 2008] D. Spinellis. Software builders. *Software, IEEE*, 25(3):22 –23, May 2008.

[Stratton *et al.*, 2007] W.C. Stratton, D.E. Sibol, M. Lindvall, and P. Costa. The save tool and process applied to ground software development at jhu/apl: An experience report on technology infusion. In *Software Engineering Workshop, 2007. SEW 2007. 31st IEEE*, pages 187 –193, February 2007.

[Strein *et al.*, 2006] Dennis Strein, Hans Kratz, and Welf Lowe. Cross-language program analysis and refactoring. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '06, pages 207–216, Washington, DC, USA, 2006. IEEE Computer Society.

[Tallon, 2010] Paul P. Tallon. Understanding the dynamics of information management costs. *Commun. ACM*, 53(5):121–125, May 2010.

[Terra and Valente, 2009] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Softw. Pract. Exper.*, 39(12):1073–1094, August 2009.

[Tichelaar *et al.*, 2000] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*, pages 157–167, Los Alamitos, CA, November 2000. IEEE Computer Society Press.

[Tilley *et al.*, 1996] Scott R. Tilley, Dennis B. Smith, and Santanu Paul. Towards a framework for program understanding. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, page 19. IEEE Computer Society, 1996.

[Tonella and Potrich, 2005] Paolo Tonella and Alessandra Potrich, editors. *Reverse Engineering of Object Oriented Code*. Springer, 2005.

[Tzerpos and Holt, 2000] V. Tzerpos and R.C. Holt. Accd: an algorithm for comprehension-driven clustering. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 258 –267, 2000.

[Ulrich and Newcomb, 2010] William M. Ulrich and Philip Newcomb. *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[Vanciu and Rajlich, 2010] R. Vanciu and V. Rajlich. Hidden dependencies in software systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1 –10, September 2010.

[Viscusi *et al.*, 2010] Gianluigi Viscusi, Carlo Batini, and Massimo Mecella. *Information Systems for eGovernment - A Quality-of-Service Perspective*. Springer, 2010.

[Walker *et al.*, 2006] Robert J. Walker, Reid Holmes, Ian Hedgeland, Puneet Kapur, and Andrew Smith. A lightweight approach to technical risk estimation via probabilistic impact analysis. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 98–104, New York, NY, USA, 2006. ACM.

[Willmor *et al.*, 2004] D. Willmor, S.M. Embury, and Jianhua Shao. Program slicing in the presence of database state. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 448 – 452, September 2004.

[Xiao and Tzerpos, 2005] Chenchen Xiao and V. Tzerpos. Software clustering based on dynamic dependencies. In *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, pages 124 – 133, March 2005.

[Xu *et al.*, 2005] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, March 2005.

[Yazdanshenas and Moonen, 2011] Amir Reza Yazdanshenas and Leon Moonen. Crossing the boundaries while analyzing heterogeneous component-based software systems. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 193–202, Washington, DC, USA, 2011. IEEE Computer Society.

[Yazdanshenas and Moonen, 2012a] Amir Reza Yazdanshenas and Leon Moonen. Fine-grained change impact analysis for component-based product families. In *International Conference on Software Maintenance (ICSM)*. IEEE, September 2012.

[Yazdanshenas and Moonen, 2012b] A.R. Yazdanshenas and L. Moonen. Tracking and visualizing information flow in component-based systems. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 143 –152, June 2012.

[Ying *et al.*, 2004] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll. Predicting source code changes by mining change history. *Software Engineering, IEEE Transactions on*, 30(9):574 – 586, September 2004.

[Zhifeng Yu, 2001] Václav Rajlich Zhifeng Yu. Hidden dependencies in program comprehension and change propagation. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 293–299, Washington, DC, USA, 2001. IEEE Computer Society.

[Zimmermann *et al.*, 2004] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press.

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name	Fabrizio Perin
Matrikelnummer	09–133–521
Studiengang	Dissertation in Computer Science
Titel der Arbeit	Reverse Engineering Heterogeneous Applications
Leiter der Arbeit	Prof. Dr. Oscar Nierstrasz

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe o des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Fabrizio Perin
Bern, 22. November 2012

Curriculum Vitae

Personal Information

Name	Fabrizio Perin
Date of Birth	April 9, 1981
Place of Birth	Sesto S. Giovanni, Italy
Nationality	Italian

Education

2008 – 2012	Ph.D. in Computer Science at the Software Composition Group, University of Bern, Switzerland. Thesis title: *Reverse Engineering Heterogeneous Applications* .
2004 – 2007	Master in Computer Science at the Dipartimento di Informatica DISCo, Università degli Studi di Milano Bicocca, Milan; Italy. Thesis title: *Dynamic analysis for Design Pattern detection in Java: Information collected using JPDA*.
2000 – 2004	Bachelor in Computer Science at the Dipartimento di Informatica DISCo, Università degli Studi di Milano Bicocca, Milan; Italy.

Complete Curriculum Vitae:

`http://www.linkedin.com/pub/fabrizio-perin/2a/181/96b`